CONTROL YOUR TRS-80

Better BASIC and Machine Code

IAN STEWART · ROBIN JONES · NEVIN B. SCRIMSHAW

Control Your TRS-80

Better BASIC and Machine Code

Control Your TRS-80

Better BASIC and Machine Code

Ian Stewart
Robin Jones
and Nevin B. Scrimshaw

Birkhäuser
Boston • Basel • Stuttgart

Library of Congress Cataloging in Publication Data

Stewart, Ian.
Control your TRS-80.

Bibliography: p.
Includes index.
1. TRS-80 (Computer) – Programming. I. Jones, Robin.
II. Scrimshaw, Nevin, 1950- . III. Title.
QA76.8.T18S74 1983 001.64'2 83-12221
ISBN 0-8176-3143-7

ISBN 0-8176-3143-7
Printed in USA
A B C D E F G H I J

The Rubáiyát of Programmer Khayyám

Awake! For Morning's fickle hand doth load
Updated software in the daylight mode.
    Return from sluggish subroutine of night:
DIM the array, but brilliant the code!

Myself when young did frequently frequent
The data-punching rooms, and heard great argument;
    But evermore it seemed I must emerge
By that same interface wherein I'd went.

Ah, but my computations, people say,
Process the text to clearer meaning? Nay,
    Though Man may seek the symbols to construe
The Greater Editor will have his way.

The User programs while the disk-drives whisk;
Taps the mad keyboard of a mind at risk.
    The work of years comes suddenly to naught
As random noise corrupts the floppy disk.

Some for the glories of this world, and some
Sigh for a pointer to the world to come.
    Ah, seize the output, let the record go,
Nor heed the rumble of magnetic drum!

A User-Manual 'neath a labelled tree,
A pint of beer, a ploughman's lunch—and Thee!
    What care I then for megabytes?
Thy tiniest bits yield megabytes for me.

The moving cursor writes, and having writ
Moves on: nor all your piety nor wit
    Shall lure it back to cancel half a line
Nor all your Tears wash out a word of it.

But wait! say ye: The console's cursor keys
Can Backspace, Delete, Edit as we please?
    Not so! These merely tidy the display:
Still the grim input's in the memories.

Some peek the ROM of Time's predestined flight;
Some seek within Life's RAM new lines to write.
    In vain each strives t'assemble faultless code,
For still Death's Digits poke the final byte.

Contents

Preface

This book is intended for user's of the TRS-80 Model's III and IV who have mastered the fundamentals of BASIC and wish to delve a little deeper. Its main objectives are to develop techniques for writing more complicated programs clearly and rapidly; to describe a variety of useful data structures for holding information; and to introduce the delights of Machine Code.

The principles of good programming are machine independent and almost everything in this book applies (subject to important changes in the system variables) to any machine built around the Z80 or Z80A CPU. In particular, users of the Model IV will enjoy a well spent tour of the Model III that coexists inside their computer.

This book is for those who have mastered the basics, and want to cut their teeth on some thing more substantial.

One thing you can do is write more complicated BASIC programs—after all, that's why high level-languages were invented. To write bug-free programs which can be understood and modified without courting the lunatic asylum, you need to develop a well structured style. We pursue this line of thought in the sections on *Data Structures* and *Structured Programming*.

You can also go outside BASIC altogether. On the TRS-80 that means *Machine code*: the language that the Z80 microchip talks. Often (though not always) Machine Code is faster than BASIC: the snag is that you have to do more of the thinking yourself by way of compensation. We didn't think that you would want to write *long* Machine Code routines at this stage; but we've given enough information for you to take several steps along that road, so more advanced books become accessible.

One important feature of Machine Code is that it teaches you a lot more about the way the computer actually works—and in the coming

age of microeverythings, that's going to be an important piece of knowledge.

We've included plenty of programs as examples: searching a stock list, enqueing and dequeing data, pushing on to a stack and popping from one, cataloging a library, a game tree where the computer works out its own winning strategy, a simple word processor, a simulation study of supermarket checkouts which applies equally well to allocating hospital beds, an educational software package that tests your French vocabulary.

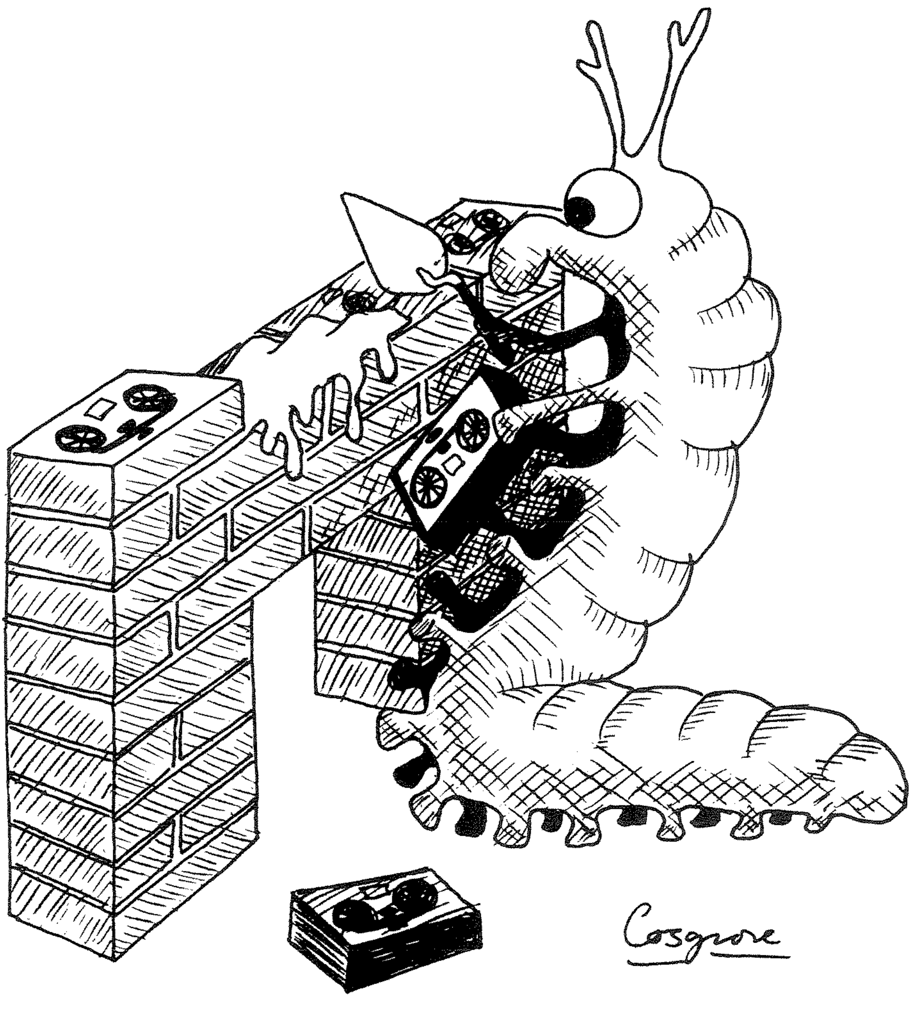
There are Machine Code routines to give the display a checkerboard pattern (instantly), to draw Hexmas trees, to add and multiply numbers, to move data around in RAM, to scroll individual regions of the screen fast enough to be useful, to scroll sideways, and to renumber BASIC lines.

The appendices include several tables of useful data: hex/decimal conversion (including 2's complement notation for relative jumps); a summary of Z80 commands; a list of all 694 Z80 opcodes. In short, all you need to control your TRS-80!

Acknowledgements

Special thanks are due to the following people, without whose help this book would not have been written: Klaus Peters, for his vision; Mary Fitz-Gibbon, for her patience; Cort Shurtleff, the long program sleuth; David Epstein, for shooting troubles; Tad Gaither, for service beyond the call of duty; Adrien Stair, the first steps, and finally, Arts & Letters, for their patience, good cheer and talent.

Data Structures



There's a lot more to programming than just knowing the features of a particular programming language. For one thing, we need a clear idea of the procedure, or *algorithm*, we are going to employ in solving a given problem. We need to develop ways of splitting this algorithm up into chunks small enough to be convenient for coding in BASIC, or whatever language we're using. That's a topic we'll come back to. For the minute, I'm concerned about an even more fundamental problem in program design: that of the organization, or structure, of the data which the program is to manipulate. Actually, we're quite used to the idea of imposing a structure on data which we wish to handle manually, although we probably wouldn't give the operation such a grand title. Think about a bank statement for example:

Date	Details	Debits	Credits	Balance
7/3	B/FWD	–	–	148.78 Ø
7/5	897669	24.48	–	124.30 Ø
7/9	S.TILL	–	20.00	104.30 Ø
8/1	W.L.INC.	–	254.11	358.41

It's clear that the data presented here are structured in a very formal way. Each type of data – date, debit, credit and so on – only appears in a given column. There's even some implied organization – we know, without being told, that the first number in the date is the month, and the second is the day. (This implicit structure can get us into trouble – if the date is read by someone from Great Britain he'll see it the other way round!) Of course, somebody had to decide in the first place that this was a convenient way to present bank statements and, naturally, he would have asked the question: "How easy is it going to be to deduce useful information from this structure?" We, on the other hand, are not interested in deriving information directly from a data structure, but in getting the computer to derive the information for us. So we are always going to have to ask the question: "how easy is it going to be to write programs to derive useful information from a particular data structure?"

So it's clear, I think, that it's important to have the data organized before you start to think about the program.

We're going to look at a number of common data structures in this book, and to describe some of the ways in which they are used. The list isn't exhaustive, and in some cases it can be useful to design a structure which is quite novel; so don't be put off from doing so simply because a structure which seems convenient isn't listed here!

Arrays are directly available in BASIC and easy to use: for example, to list commercial data. But there are other ways to use them—including the analysis of weather-maps.

1 Arrays

Some structures are built into languages, and some you have to build for yourself. In BASIC, there is only one data structure within the language. It's called an *array*, and it seems a convenient place to start. Remember, on the TRS-80 only the first two letters in a variable name are significant.

Let's review the way we think about the organization of the computer's memory. We could see it as a series:

R3	7
ZCOUNT	4Ø
ABLE	1.8
P$	ABC
FRED	−15
LETTER$	PQR5
JIM	Ø

of cells, each able to contain a number or string of characters, and each given a name, as in the diagram, for example.

This is fine for many applications, but not for all. For instance, suppose we want to input 2ØØ numbers and hold all their values at the same time. It's no good writing:

```
1Ø  FOR P = 1 TO 2ØØ
2Ø  INPUT X
3Ø  NEXT P
```

because with every loop, a new value will be entered into X, wiping out

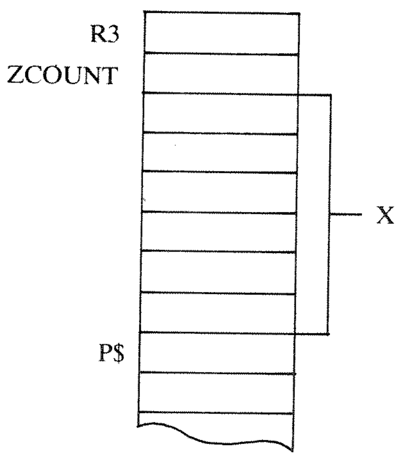
the old one. So, in the end, only the final value in the sequence will be in memory.

We could write:

```
  10  INPUT X1
  20  INPUT X2
  30  INPUT X3
  40  INPUT X4
  . . . . . . . . . . .
2000  INPUT X200
```

but it would be a pain, and anyway this will occupy a lot of memory within the program.

An array provides us with a way out of the dilemma. BASIC allows us to specify a whole block of memory as having just one name:



The block of memory represented above is called the *array* X. (It can have any valid BASIC name; see page 100 of the reference manual.)

An array may have any number of memory cells in it, but we have to tell the interpreter, at the beginning of the program, how many cells to allocate to it. This is done with a DIM (short for dimension) statement. In the above case, X has 6 cells, so we would write:

```
10  DIM X(6)
```

We still need to be able to refer to individual cells, or *elements,* within X. BASIC allows us to talk about X(1), the first element of X; X(2), the second; X(3), the third; and so on.

Coming back to our original problem, we could write:

```
10  DIM X(200)
20  INPUT X(1)
30  INPUT X(2)
40  INPUT X(3)
```

"Hang on!" you're all saying. "That's no better than it was before. In fact, it's worse, because there are loads of extra brackets."

True, true. However, the trick is that when we wrote X1, X2, X3 etc., the 1, 2, and 3 are part of the variable names and can't be altered; but if we write, for instance, X(P) then the computer sees this as X(1) if P = 1. X(2) if P = 2 and so on. So the value in the brackets (called a *subscript*) can be changed.

In this case, we want to change the subscript by starting it at 1, and adding 1 to it until it reaches 200, and that's a clear cue for a FOR-NEXT loop.

```
10  DIM X(200)
20  FOR P = 1 TO 200
30  INPUT X(P)
40  NEXT P
```

Why should we want to store 200 numbers all at once? About the simplest program I can think of in which it's clearly necessary, is one to print in reverse order the numbers input to it:

```
10  CLS
20  DIM X(200)
30  FOR P = 1 TO 200
40  INPUT X(P)
50  NEXT P
60  FOR P = 200 TO 1 STEP -1
70  PRINT X(P);
80  NEXT P
```

Obviously we can't print anything until the last number has been read, and we must remember all the previous numbers, since they are to be printed subsequently.

CALCULATING DISCOUNTS

Here's another example of a rather different nature.

Suppose we run a wholesaling business and we split our customers

into the following groups, each of which is offered a different discount, shown in brackets:

1.	Private	(0%)
2.	Local authority	(4.0%)
3.	Education	(6.5%)
4.	Government	(7.0%)
5.	Trade	(9.0%)
6.	Trade—special contract	(12.0%)

What we would like is a program that will accept a customer's name, his type (1-6), the number of items he's buying and the retail value per item, and print out the details of his bill.

Since the calculations depend on the customer type we might expect to have a series of IF statements like:

```
50  IF TYPE = 1 THEN...
60  IF TYPE = 2 THEN...
70  IF TYPE = 3 THEN...
```

This would be tedious but not beyond reasonable bounds. But what if there were 300 categories?...

Let's set up an array called DP (for Discount Percentage) like this:

	DP
DP(1)	0
DP(2)	4
DP(3)	6.5
DP(4)	7
DP(5)	9
DP(6)	12

Now if TYPE = 1, the discount value we're interested in is in DP(1). If TYPE = 2, the value we want is in DP(2). In other words the value we are after is always in DP(TYPE)! So the code to input the data and evaluate the discount would look something like:

```
80  INPUT N$
82  INPUT TYPE
84  INPUT NUMBER
86  INPUT PRICE
90  LET AMOUNT = NUMBER*PRICE
100 LET DISCOUNT = AMOUNT*DP(TYPE)/ 100
```

No IFs anywhere!

Note that we're using TYPE to point to the right value in the array. We call a variable used in this way a *pointer*.

How can we set up the array in the first place? The simplest way is to have a series of assignment statements like:

```
 5  DIM DP(6)
10  LET DP(1) = 0
15  LET DP(2) = 4
20  LET DP(3) = 6.5
25  LET DP(4) = 7
30  LET DP(5) = 9
35  LET DP(6) = 12
```

This is OK for small arrays but tedious for large ones. An alternative is to use an input loop:

```
 5  DIM DP(6)
10  FOR P = 1 TO 6
20  INPUT DP(P)
30  NEXT P
```

but, of course, we don't want to execute this routine every time the program is run, although we may wish to do so occasionally, if the discounts change. So we could have:

```
1  CLS
2  INPUT "NEW DISCOUNTS (Y/N)"; A$
3  IF A$ = "N" THEN 80
```

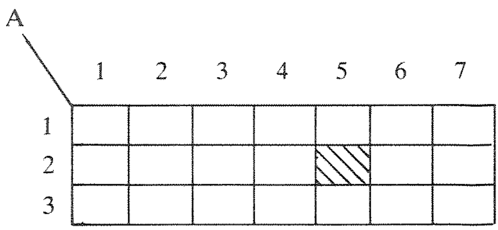
so that the program jumps round the input routine if the discounts aren't changed. When you save the program, the array values are also saved, so this will work for successive runs provided you key in GOTO 1 and not RUN (RUN clears all variable values).

ARRAYS IN TWO DIMENSIONS

What I've been describing so far is called a *one-dimensional array* or *vector*. It's possible to have a *two-dimensional array* or *table*. We specify such an array by a DIM statement as before. For example:

```
10  DIM A(3,7)
```

specifies an array which looks like:



In other words, I've identified for the computer a table having 7 columns and 3 rows. There isn't, though, any standard convention which decrees that the number of columns must appear before the number of rows in the DIM statement. It's the way you think about the table that matters. In other words, I could, equally validly, have specified the same array by:

```
10  DIM A(7,3)
```

provided that in each subsequent reference to the array, I always give the row number second, and the column number first.

The golden rule is: having chosen a convention, stick to it; you're less likely to make a mistake. In this book I'll use what is probably the commonest convention: give a row number first, the column number second. That's convenient because one-dimensional arrays are usually drawn as single columns.

Now, when we want to refer to a particular cell in the table, A, we can do so by specifying the row and column which intersect at that cell; so if we wanted to set the shaded cell in the diagram to 24, we could write:

```
50  LET A(2,5) = 24
```

OK; that's the basic idea.

Now let's put it to some use.

Imagine that you are a geographer who wants to keep a record of the annual rainfall in a particular region. There are weather stations dotted around the region, so we might have a map which looks like Figure 1.1, in which the region of interest is shown shaded, and a star denotes a weather station.

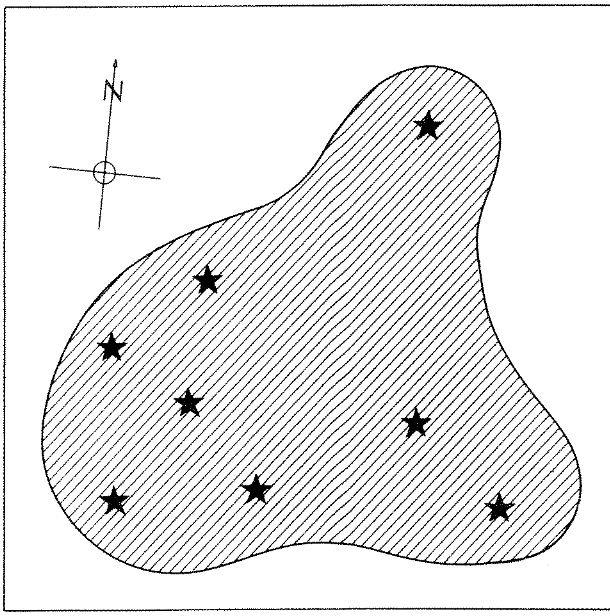


Figure 1.1

Obviously, the map is two dimensional, and we can easily represent it in a corresponding two-dimensional array by giving each point on the map a numeric code. There are only three types of point:

1. Points inside the region not at a weather station.
2. Weather stations.
3. Points outside the region.

At the weather stations we know the rainfall so the corresponding position in the array can just contain this value (in millimeters). Two values which can't be confused with a rainfall value are now needed to identify points inside and outside the region. Since you can't have negative rainfall, we could choose −2 for a point outside the region and −1 for one inside. So an array which models the map would look something like:

−2	−2	−2	−2	−2	−2	−2	−2
−2	−2	−2	−2	−2	461	−2	−2
−2	−2	−2	−2	−1	−1	−2	−2
−2	−2	4Ø1	−1	−1	−1	−2	−2
−2	4ØØ	−1	−1	−1	−1	−2	−2
−2	−1	44Ø	−1	−1	48Ø	−2	−2
−2	42Ø	−1	424	−1	−1	484	−2
−2	−2	−2	−2	−2	−2	−2	−2

Of course, the resulting shape doesn't look as smooth as the original. However, we can always improve the appearance (and accuracy) of the representation by making the array bigger.

OK, we've managed to store a representation of a map in the computer. So what? What's wrong with a conventional atlas? Nothing, except that a conventional atlas leaves you to deduce whatever information you need from it. Now that our representation is inside the computer's memory we can write programs to answer all sorts of questions about the map:

How many weather stations are there?

What percentage of them have rainfall figures over 41Ø mm?

What is the highest recorded rainfall, and where is it?

What is the area of the region?

With a map of practical size, it would be very tedious to answer any of these questions manually; let's see how simple it would be to write the programs.

We'll suppose that the array has been set up by:

```
10  DIM MAP(5Ø,5Ø)
```

and that the values have already been input to it. We want to know how many stations there are.

So a rough outline for the procedure would be:

- Examine a cell.
- If it contains a non-negative value it's a weather station so count it.
- Repeat the process for all cells.

For a particular cell, somewhere in MAP, whose column value is C and whose row value is R, the IF statement we need is:

```
1Ø3Ø  IF MAP(R,C) > = Ø THEN LET NWS = NWS + 1
```

[NWS = number of weather stations]

To deal with all the columns from 1 to 5Ø in a row R, we need a FOR loop:

```
1Ø2Ø  FOR C = 1 TO 5Ø
1Ø3Ø  IF MAP(R,C) > = Ø THEN LET NWS = NWS + 1
1Ø4Ø  NEXT C
```

and to deal with all the rows from 1 to 5Ø, we need a FOR loop round that:

```
1Ø1Ø  FOR R = 1 TO 5Ø
1Ø2Ø  FOR C = 1 TO 5Ø
1Ø3Ø  IF MAP(R,C) > = Ø THEN LET NWS = NWS + 1
1Ø4Ø  NEXT C
1Ø5Ø  NEXT R
```

Now all we need to do is make sure that NWS contains zero to begin with:

```
1ØØØ  LET NWS = Ø
```

and print the result out at the end:

```
1Ø6Ø  PRINT "NO. OF STATIONS = "; NWS
```

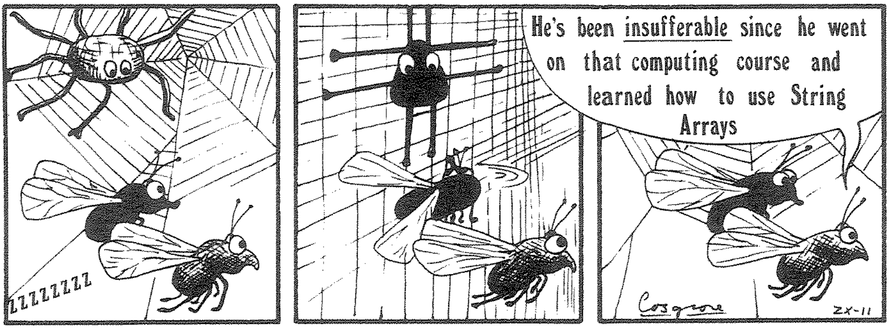
The other three problems I listed are very similar. They all require the same pair of FOR loops. The IF statements, and what to do when they are true, change, of course. You might like to try them.

ARRAYS IN THREE DIMENSIONS

Let's suppose that our geographer wants to use a bit more raw data. He's now got rainfall figures for each month at each weather station.

So we need twelve two-dimensional arrays like MAP to represent these data. Each array is like a page of an atlas. Shouldn't we be able to combine the arrays, like binding the atlas pages into a book? Well, we can. What we end up with is a three-dimensional array, which we could define by:

```
1Ø  DIM BIGMAP(5Ø,5Ø,12)
```



(remember the TRS-80 thinks of BIGMAP as BI) and the third value in the brackets represents the number of months. On the TRS-80 you have to use the CLEAR command to allocate memory space for a large array. In fact this array will never fit in 16K.

We can ask all the same questions as before, and the only change in the programs will be that we need another, outside, loop to change the month value.

For instance, the new loop could be:

```
FOR MONTH = 1 TO 12
..........
NEXT MONTH
```

and the IF statement will look like:

```
IF BIGMAP(R,C,MONTH) = ... THEN ...
```

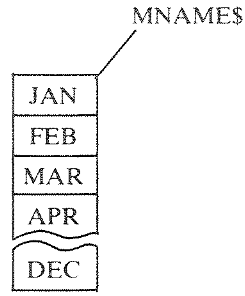
(Of course, some questions, like "How many stations are there?" don't change with time, and if you try this with that program you'll just get 12 times as many stations as you should.)

How about a routine to tell us the rainfall figure for a particular station in a given month? The code could look like this:

```
2000 PRINT "ENTER MAP REFERENCE OF STATION (R,C)"
2010 INPUT R
2020 INPUT C
2030 PRINT "ENTER 1ST 3 LETTERS OF MONTH"
2040 INPUT M$
2050 IF M$ = "JAN" THEN LET MNTH = 1
2060 IF M$ = "FEB" THEN LET MNTH = 2
2070 IF M$ = "MAR" THEN LET MNTH = 3
.............
and so on
.............
2160 IF M$ = "DEC" THEN LET MNTH = 12
2170 PRINT "REQUIRED FIGURE IS:" ;BIGMAP(R,C,MNTH); "MM"
```

Lots of IFs again! How can the chunk of code between lines 2050 and 2160 be improved?

Let's set up an array of month names (much as we setup the discount table earlier). We'll call it MNAMES.



and then replace the relevant lines with:

```
2050  FOR MNTH = 1 TO 12
2060  IF MNAME$(MNTH) = M$ THEN 2170
2070  NEXT MNTH
```

Get it? It searches through the MNAMES array looking for a match with the input month (M$). When it finds one, the value of the pointer, MNTH, is the numeric equivalent. A good maxim is:

> If you've just written a piece of code with a lot of similar IF statements, one after the other, there was probably an easier way.

ARRAYS IN HIGHER DIMENSIONS

Perhaps you noticed that I carefully refrained from saying "A three-dimensional array looks like a cube" in the preceding section. Of course, you can think about it like that, but it makes it difficult to think about what a four-dimensional array (yes, you can have one!) behaves like, because it doesn't *look* like anything. We can't imagine four-dimensional solids. But that doesn't imply that it can't mean something.

At the risk of flogging a dead geographer, let's return to our map problem, and suppose that now he provides us with the monthly figures for a decade! So we now have ten 3D arrays. Why not call that one 4D array, just as we called the twelve 2D arrays one 3D array. Now we refer to a particular cell in the array as:

```
EVENBIGGERMAP(R,C,MONTH,YEAR)
```

and we only have to provide the appropriate values of R, C, MONTH and YEAR to elicit the required data.

In principle, there's nothing to stop us having five-, six-, seven- or even higher-dimensional arrays. Each extra dimension is used to store

another attribute such as population, death rate, temperature and so on.

In practice, we're limited by the available memory. Actually, as far as 16K is concerned, we had already run out of memory at the 3D stage. It's easy to calculate:

We had a 5Ø × 5Ø × 12 = 3Ø,ØØØ cell array.

Each cell occupies 2-8 bytes, so we are using at a minimum 6Ø,ØØØ bytes. A 16K memory has 16 × 1Ø24 = 16,384 bytes. Oops! (see page 221 of your reference manual for the details.)

Of course, we can always make the map smaller to fit in; 16 × 16 × 12 just fits (but by the time the program is stored as well, it might not!). So these refinements are really for larger machines, although there can be occasions when a 7 × 7 × 7 × 7 four dimensional array (which is getting on toward the largest possible in 16K) is useful.

Given a list of data, how do you find the item you want?
You *could* just read through the whole list . . .
but there may be better ways.

2 Searches

It's obvious that one of the most useful things we can do with an array is to search it for some particular piece of information; in fact, we've already done so, in our examples, several times. On each previous occasion, we've shown what is known as a *linear search,* so called because every element of the array is examined in turn until the target cell is reached. This is fine for small arrays, but with large ones it can be rather time-consuming.

There's an alternative, known as a *binary search,* which is faster, provided the data are in a known order in the array. Let's look at an example. Remember my wholesaler with his discount system? He's now getting ambitious. He wants to implement a stores control system. Each item he deals with is given a reference number, and the information held about each item is the number in stock and the price per unit.

So we could set up an array like:

STOCK

Ref. No.	No. in stock	Price
1384	58	31.72
1791	246	2.60
2114	15	254.00
2164	2486	0.53
8561	1418	0.16

Note that the reference numbers are in ascending order. The procedure won't work otherwise.

Among other routines, we would like one which allows the storeman to enter an item reference number, and the system responds by displaying the number currently in stock, and the price of the item.

The way a binary search does this is first to look at the middle row in the table. Thus if there are 100 rows, we look at the 50th. (As there isn't an exact middle row if the number of rows is even, the 51st will also work, if you prefer.) If the reference number we're looking for is greater than the one we find here, it can't be in the lower half of the table, so we discard that, and concentrate on the upper half. Now we repeat the process for the remaining part of the table. This will chop out one half of the remainder, and continued repetition will ultimately narrow the search to one row.

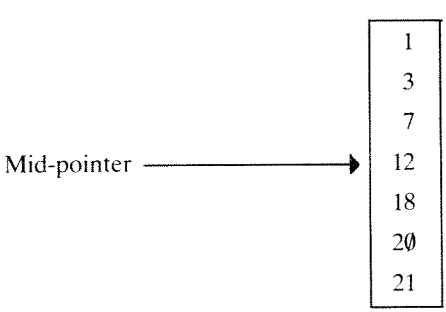
Sounds long winded? Well, think about an example. Suppose there are 1000 items in the table. According to Murphy's law the one you're after is always the last one you look at, so a linear search requires an examination of all 1000 items. A binary search removes:

500	items	from	consideration	on	the	1st	test
250	"	"	"	"	"	2nd	"
125	"	"	"	"	"	3rd	"
62	"	"	"	"	"	4th	"
31	"	"	"	"	"	5th	"
15	"	"	"	"	"	6th	"
7	"	"	"	"	"	7th	"
3	"	"	"	"	"	8th	"
1	"	"	"	"	"	9th	"

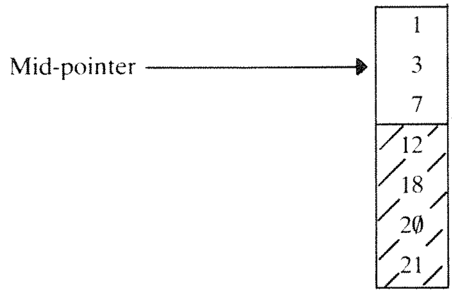
which means we must find the target on the 10th test (even with Murphy's pessimism)!

The diagram below may make the idea a little clearer.

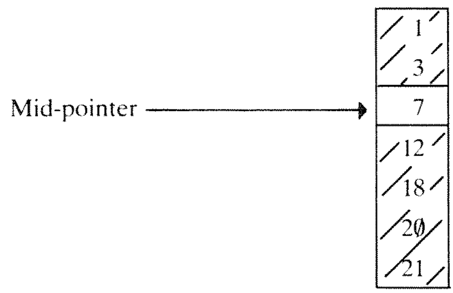
Target:7



12 > 7, so ignore the top half of the table.



3 < 7, so ignore the bottom half of the table.



Gotcha!

Let's write a routine to do this for the STOCK array. Since STOCK is an illegal variable name we shall refer to it as STCK in our program. We need three pointers; one to the top of the region we're examining (TP), one to the bottom (BP), and the mid-pointer (MP), which is just (BP + TP)/2. So for an array with 1000 rows we start with:

```
2500  LET BP = 1
2510  LET TP = 1000
2520  LET MP = INT((BP+TP)/2)
```

The INT is strictly necessary because otherwise MP may include a decimal part (as here: 1001/2 = 500.5) which doesn't make sense.

Now we need to compare the required reference number (assuming this has already been entered into RN) with the reference number in STOCK pointed at by MP:

```
2530  IF STCK(MP,1) = RN THEN GOTO 3000
```

(Remember that item reference numbers are all in column 1).

So when we get to line 3000, MP is pointing to the row we want and we'll be able to write:

```
3000  PRINT "NO. IN STOCK IS"; STCK(MP,2)
3010  PRINT "PRICES"; STCK(MP,3)
```

If the condition isn't met, however, we need to know if the value pointed to by the mid-pointer is greater than the target value and, if so, we chop out the top half of the table:

```
2540  IF STCK(MP,1) > RN THEN LET TP = MP - 1
```

On the other hand, the value pointed to by MP might be less than the target value:

```
2550  IF STCK(MP,1) < RN THEN LET BP = MP + 1
```

Now we need to calculate the new mid-pointer so:

```
2560  GOTO 2520
```

Tying the whole thing together gives:

```
2500  LET BP = 1
2510  LET TP = 1000
2520  LET MP = INT((BP + TP)/2)
2530  IF STCK(MP,1) = RN THEN GOTO 3000
2540  IF STCK(MP,1) > RN THEN LET TP = MP - 1
2550  IF STCK(MP,1) < RN THEN LET BP = MP + 1
2560  GOTO 2520
3000  PRINT "NO.IN STOCK IS"; STCK(MP,2)
3010  PRINT "PRICE IS"; STCK(MP,3)
```

Of course, there are lots of refinements you could build in. What happens if you enter a nonexistent item reference number, for instance? I'll leave you to think about that one.

"Last in, first out" is the rule for a stack, either in the Tower of Hanoi or in a machine code program. Here's how to build an efficient one.

3 Stacks

As I've said before, arrays are the only data structures inside BASIC. Other structures have to be built, usually in terms of arrays. So let's look first at a structure which is very simple to implement in terms of an array, called a *stack*.

The name "stack" explains its operation pretty well. You can only pile things on top of it, and you can only remove items from the top.

For example, we might start with:

```
| 4 |
| 2 |
| 7 |
+---+
```

If we add 9 and 12 (in that order) to the stack we get:

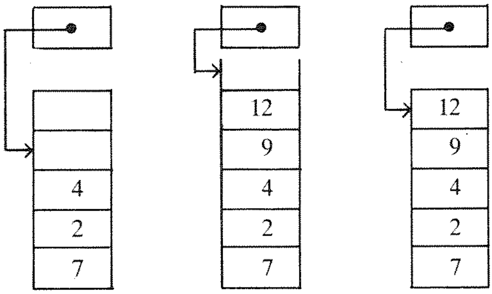
```
| 12 |
|  9 |
|  4 |
|  2 |
|  7 |
+----+
```

Remove an item from the stack and we have:

```
| 9 |
| 4 |
| 2 |
| 7 |
+---+
```

Obviously what we have is very like a one-dimensional array, but with the added restriction that access may only be made at a particular place (the top of the stack). Unfortunately, the top moves about inside the array depending on how many items are on the stack. So we need a pointer to determine where the top of the stack is. We'll define it to point at the first free location on the stack.

So now the previous example looks like:



Notice that when the 12 is removed from the stack it does not have to disappear from the array, because the pointer tells us where the top is to be considered to be.

STACK ROUTINES

OK. Let's try writing BASIC routines to implement a stack. First of all, what routines are needed? Well, there are three:

1. Set up the stack in the first place. We'll call this INITIALIZE.
2. Load a value on to the stack. The technical term for this is PUSH.
3. Remove a value from the stack. This is called POP.

The INITIALIZE routine can be put right at the beginning of the program:

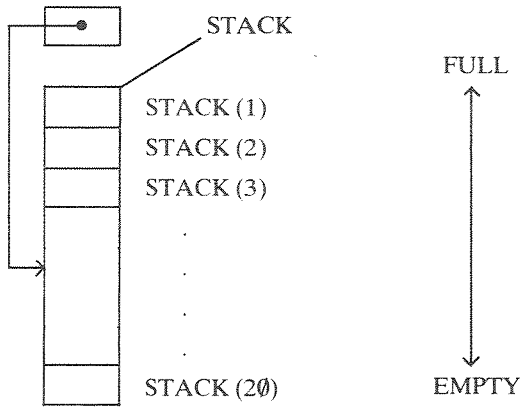
```
10  CLS
20  DIM STACK(20)
30  LET SP = 20
```

For a stack to hold a maximum of 20 items that's all it is! Putting the stack pointer (SP) to 20 tells us the stack is empty because it defines STACK (20) to be the first free location.

The other two routines will be needed throughout the main program and so they will be subroutines. PUSH looks like:

```
5010  LET STACK(SP) = V
5020  LET SP = SP - 1
5030  RETURN
```

(assuming that V contained the value to be PUSHed). What's happening is that the value is put into the array where SP is pointing. As that is the first free location, that's fine. Then the stack pointer is moved to point to the next free location. One is subtracted from it because of the mental model I have of the stack:



(There's nothing to stop you thinking about it the other way round if you prefer; provided, of course, you do it consistently.)

Actually, this diagram gives us a clue to something we haven't considered: what happens if the array fills up? SP will contain zero, and an attempt to execute line 5010 will lead to an error message. We need a line 5000 to test for this condition:

```
5000  IF SP = 0 THEN 5040
```

and:

```
5040  PRINT "STACK FULL"
5050  END
```

POP will be similar:

```
6000  IF SP = 20 THEN 6040
6010  LET SP = SP + 1
6020  LET V = STACK(SP)
6030  RETURN
6040  PRINT "STACK EMPTY"
6050  END
```

EXAMPLE: THE TOWER OF HANOI

The "Tower of Hanoi" puzzle provides a simple example of the use of stacks. In its initial state, the puzzle looks like this:

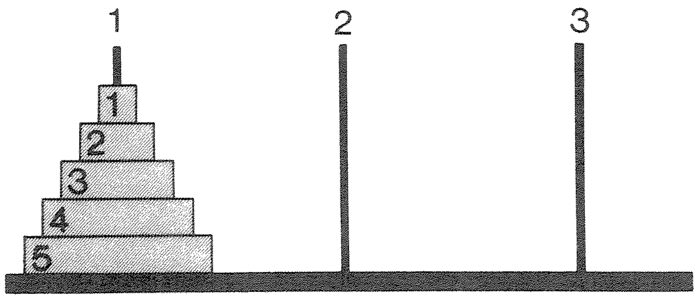


Figure 3.1

There are five discs with holes drilled through their centers, mounted on a spindle, with the largest disc at the bottom and the smallest at the top as shown in Figure 3.1. A disc may be removed from the top of this heap (or any other that forms) and placed on one of the other two spindles.

Obviously, we've got three stacks. But there's an extra constraint: at no stage may a larger disc sit on top of a smaller one. The goal is to transfer the whole tower to one of the other spindles.

Our problem is to allow a player to make moves, and display the state of the discs at each stage, checking for illegal moves.

Since there are three stacks we need to make some modifications. We'll have a pointer P to the current stack, so that line 2Ø becomes

```
2Ø   DIM STACK(2Ø,3)
```

Also we need three separate stack pointers, which we'll put in an array called SP:

```
15   DIM SP(3)
2Ø   LET SP(1) = 2Ø
21   LET SP(2) = 2Ø
22   LET SP(3) = 2Ø
```

(Incidentally, we're only using 2Ø here because it was used before: a smaller stack would be fine.) Now all references in PUSH and POP to STACK(SP) become references to STACK(SP(P),P). For instance, line 5Ø1Ø reads:

```
5010 LET STACK(SP(P),P) = V
```

and 5020 becomes

```
5020 LET SP(P) = SP(P) - 1
```

Now, before we call PUSH or POP, we have to ensure that P is set to the right stack.

Here goes. The first job is to set up the left-hand stack (P = 1) to contain the discs. We can identify these by the numbers used in Figure 3.1. So:

```
100 LET P = 1
110 FOR V = 5 TO 1 STEP -1
120 GOSUB 5000
130 NEXT V
```

Then we find out which stack the player wishes to remove a disc from:

```
140 INPUT "WHICH PILE TO BE REDUCED?"; SR
```

and we unstack that value

```
150 LET P = SR
160 GOSUB 6000
```

At this stage the required value is in V. We don't need to test whether there was a disc in the stack to remove, because POP tests for an empty stack anyway! Now we ask where the disc is to be put:

```
170 INPUT "WHICH PILE TO BE INCREASED?"; SI
```

Now see if the move is legal. This is going to need some thinking about, so we'll put off the day of reckoning and simply define a subroutine, from which a return only occurs if the move is valid:

```
180 GOSUB 300
```

So if 190 is reached the move was legal, and we can PUSH into the stack SI, and then repeat the whole moving experience...

```
190 LET P = SI
200 GOSUB 5000
210 GOTO 140
```

It remains to attack LEGALCHECK. We need to confirm that the value on top of the SI stack is greater than the value in V. That involves

POPping this stack—which will overwrite the value in V! So we'll save V first:

```
300  REM LEGAL CHECK
310  LET VS = V
320  LET P = SI
330  GOSUB 6000
340  IF V > VS THEN 370
350  PRINT "ILLEGAL"
360  END
370  GOSUB 5000
380  LET V = VS
390  RETURN
```

Note lines 370 and 380: having popped the value off the stack to examine it we mustn't forget to push it back on, to restore its state. Also, since we moved V out into VS temporarily, we have to restore its original value, otherwise the PUSH at line 220 will be pushing the wrong thing!

And now I have to tell you that this won't work. The reason is that we haven't considered the special case that a stack is empty when something is added to it. Under these circumstances the program will never get past line 330, calling POP: it will simply object that it can't, because the stack is empty. The simplest thing would be to put "6" at the bottom of all three stacks to start with, so that any value in the range 1-5 is allowed to stack on top of it. I'll leave that (or the alternative: test if the stack is empty, and PUSH straight away) as an exercise for you.

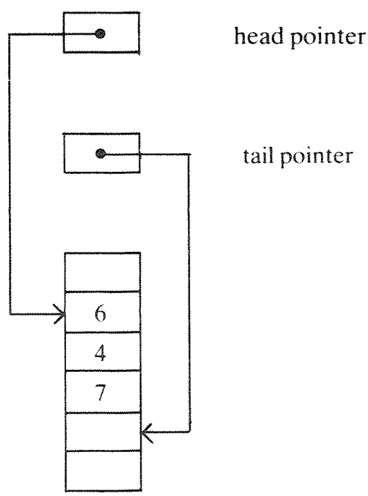
I'll also leave you with the problem of *displaying* the stacks as the game progresses. This can be as simple as printing out the internal values, if you like; or more satisfyingly, plotting each disc, using the internal value 1-5 to define the length of the line of graphics characters used.

One final comment: I've adopted a new convention here of allowing *named subroutines* (GOSUB, PUSH, etc.). Obviously that's for improved readability. You can write these names into your program listing using REM statements. This idea can also save work if you decide to change the subroutine addresses.

How to store information temporarily and process it in the order in which it arrived.

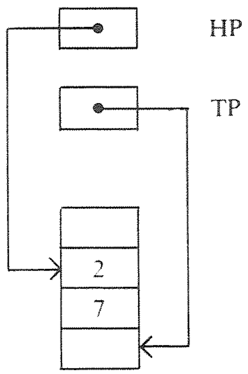
4 Queues

Again the word "queue" describes the operation of the structure well enough. Items are added to the queue at one end, but removed from the other. So now we need two pointers, one for the head of the queue and the other for the tail, like this:

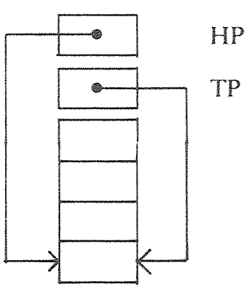


Implementing a queue is very like implementing a stack, but there's one extra problem: testing the queue to see if it's full. For instance, suppose we add two items to the queue shown above, so that the tail pointer is now beyond the array. If we didn't think too carefully about it, we could use that information to decide that the queue is full. Of course, it isn't, because there's a spare slot above the head, and if we remove items from the queue there will be more available space there, although the tail pointer hasn't moved, and it's the tail pointer we're testing to see if the queue is full.

So let's rethink the problem. Firstly, when the tail pointer falls off the bottom of the array we want it to reappear at the top, so as to use the available array space. In other words, the array becomes a circular structure, and both pointers chase each other round it. That still leaves the question of how we tell when the queue is full or empty. Let's think about a simple example:



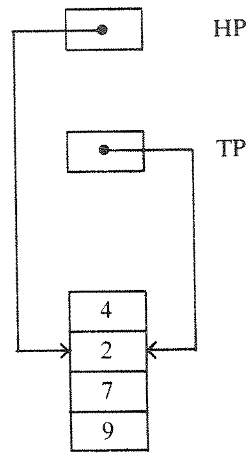
If we remove two items from this queue, we'll have to add two to the head pointer and the queue will now be empty:



(For clarity, I've removed the 2 and 7 from the array, although, as with the stack, they will actually be left there).

So there's a nice simple rule to determine when the queue is empty: the head and tail pointers are equal.

Now let's fill the original queue by adding two values (9 and 4, say) to it. So the tail pointer goes back to the beginning of the array and then gets incremented by one:



Oops! The head and tail pointers point to the same place again, so all our rule actually tells us is that the queue is either full or empty. Not much use really. . .

Never mind! We can get round it quite easily. What we've really done so far is roughly like taking a snapshot of a car race and asking somebody to look at it and say which car is in the lead. Of course, he can't, because he doesn't have any information about how many laps each car has done. We're in a similar position. The pointers could be neck and neck (empty) or the tail pointer could be about to lap the head pointer (full). So we don't need to know exactly how many laps each pointer has done, only the difference between them. This can only be zero or one, because to allow more would overfill the array.

QUEUE ROUTINES

Having dealt with the problem, the rest of the programming is pretty similar to that for a stack. Again, we need three routines, called INITIALIZE, ENQUEUE and DEQUEUE. As before, INITIALIZE can appear at the beginning of the program:

```
 5  CLS
10  DIM QUEUE(64)
20  LET HP = 1
30  LET TP = 1
40  LET LAP = 0
```

the ENQUEUE routine is:

```
2000  IF HP = TP AND LAP = 1 THEN 2060
2010  LET QUEUE(TP) = V
2020  LET TP = TP+1
2030  IF TP > 20 THEN LET LAP = LAP+1
2040  IF TP > 20 THEN LET TP = 1
2050  RETURN
2060  PRINT "QUEUE FULL"
2070  RETURN
```

A couple of things to note:

As for a stack, I'm assuming an array of length 20, and the value to be enqueued to be held in V. Line 2000 tests to see if the queue is full. Lines 2030 and 2040 deal with the problem of completing a lap. Since two actions are necessary to do this, we can either ask the same question twice (as I have) or let:

```
2030  IF TP > 20 THEN GOSUB 2080
```

perform the two actions on 2080 and 2090 and have another RETURN on 2100.

Logically they are equivalent, but if you spray too many GOSUBs and GOTOs around your code, it looks like undisciplined knitting, and becomes very difficult to read.

DEQUEUE takes the form:

```
3000  IF HP = TP AND LAP = 0 THEN 3060
3010  LET V = QUEUE(HP)
3020  LET HP = HP+1
3030  IF HP > 20 THEN LET LAP = LAP-1
3040  IF HP > 20 THEN LET HP = 1
3050  RETURN
3060  PRINT "QUEUE EMPTY"
3070  RETURN
```

See how similar the two subroutines are? There's a kind of mirror-image symmetry about them. The same was true of the PUSH and POP routines. It's a very common phenomenon in routines which do, in some sense, opposite things. The programmer with a nose for trouble will worry when symmetry like this isn't present. It's not a hard and fast rule; just a hunch that comes with experience, but useful nevertheless.

A FOOTBALL QUEUE

Here's an example of how a queue might be used.

The video-printers used by TV networks to transmit the football results, are really just a kind of long-distance typewriter. When the operator types a character, it is not immediately transmitted, but queued. The entire queue is dequeued only when the operator hits "return." That way "DENVER" will flash on to the screen as a whole word, instead of being laboriously typed D-E-N...

Here's the necessary code:

```
5 CLS
10 DIM QUEUE$(64)
20 LET HP = 1
30 LET TP = 1
40 LET LAP = 0
110 LET V$ = INKEY$
120 IF V$ = "" THEN 110
130 IF V$ = CHR$(13) THEN 160
140 GOSUB 2000
150 GOTO 110
160 FOR I = 1 TO 20
170 GOSUB 3000
180 IF EMPTY = 1 THEN 210
190 PRINT V$;
200 NEXT I
210 PRINT
220 LET EMPTY = 0
230 GOTO 110
```

There are some important modifications needed to ENQUEUE and DEQUEUE, because we're dealing with characters, not numbers. In particular V becomes V$, and the arrays become string arrays so QUEUE(1) becomes QUEUE$(1) and so on. When the queue is flushed we don't want "QUEUE EMPTY" appearing on the screen—imagine the viewers' surprise at this little-known team playing against Dallas—so we replace line 3060 of DEQUEUE by

```
3060 LET EMPTY = 1
```

Now we simply test EMPTY on the return to see if the queue is empty yet; and if it is, we start on the next word.

We are here using EMPTY as a *flag:* we wave it (i.e. set it to the value 1) to show that something interesting has happened. Then, to find out later if it did happen, we just look at the flag. If it's at Ø, then it didn't happen after all. Flags are important later in the machine code chapters.

At the moment you'll get a printout like

```
DENVER 2
MIAMI 3
```

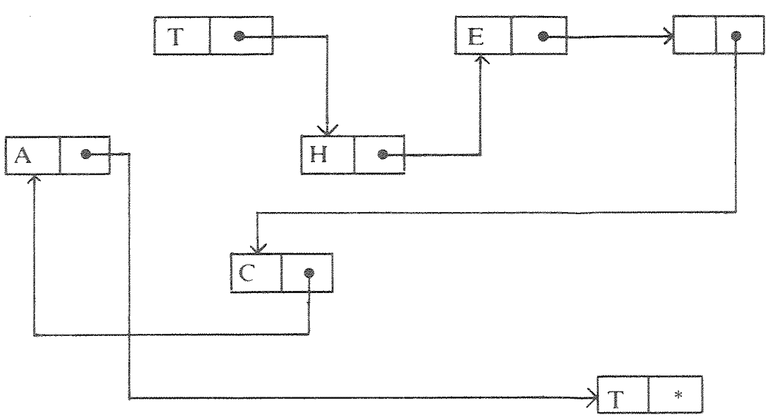
because a new line is generated at every queue-flushing operation by line 21Ø.

For another example of the use of queues see *Checkout,* Chapter 9.

The treasure-hunt principle for easily modified listings.

5 Linked Lists

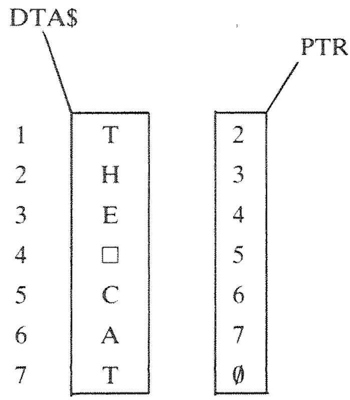
The structures we've looked at so far have all had one thing in common; if you know where a particular element is, you know where the next one is, because it's always in an adjacent memory cell. But suppose we allowed related data to be sprinkled all over memory? (Let's not worry for the moment about why we might want to.) Then we would need some explicit way of finding an item when we know where the preceding one was. The diagram below illustrates this in an abstract way.



The symbols T,E,A,H,C and "space" are arranged in no obvious pattern until you take into account that each is associated with a pointer so that provided you start at the T on the top line and follow the pointers through to the asterisk, which I'm using as a delimiter to mean "end

of list", you get the message: "THE CAT". It's the treasure-hunt principle, isn't it? Find an item, and that gives you a clue to the next.

How do we implement the structure? Well, each element has two components, some data and a pointer. So why not have a pair of corresponding one-dimensional arrays organized as shown:



so that if we want to print the contents of DTA$ in the right order we need a subroutine like:

```
1ØØØ  LET P = 1
1Ø1Ø  PRINT DTA$(P);
1Ø2Ø  IF PTR(P) = Ø THEN RETURN
1Ø3Ø  LET P = PTR(P)
1Ø4Ø  GOTO 1Ø1Ø
```

(Note that I've had to use zero, rather than asterisk, for my delimiting pointer because PTR is a number array.)

"Hang on a minute!" You're all objecting (like mad by now I expect). "You don't need to make all this fuss. You could just print out the contents of DTA$ in a FOR loop from 1 to 7."

That's right. But suppose I want to alter the message to "THE BLACK CAT". Doing it the straightforward way, I'd have to move the letters C A T down the array to make room to insert BLACK. Using the linked list all I need to do is to tag BLACK on to the end of the array and change one of the original pointers (5) like this:

	DTA$	PTR	
1	T	2	
2	H	3	
3	E	4	
4	□	8	change
5	C	6	
6	A	7	
7	T	Ø	
8	B	9	
9	L	10	
10	A	11	
11	C	12	
12	K	13	
13	□	5	

OK, it wouldn't have been a great problem to move three letters five places down the array, but suppose these were the first words of a five thousand word essay?

What we've just done is to edit a piece of text. That's the main job of what have come to be called "word-processing" programs, and I've just introduced the fundamental data structure on which most word-processors depend. I don't want to take this example any further just now, because the word-processor case study (Chapter 8) deals with it in some depth.

A LIBRARY CATALOGUE

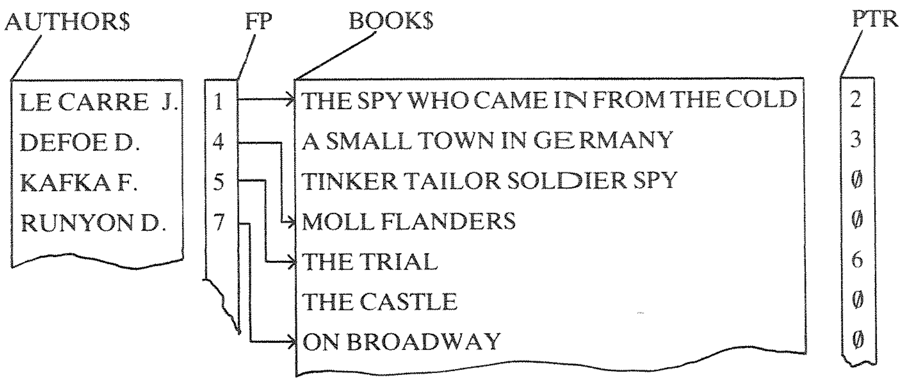
However, the linked list is such a useful structure that it's worth looking at another application for it. Let's suppose that a librarian wants to keep an author index so that when a subscriber asks "have you got any other books by Austin P. Goatwhirler?" he can answer immediately, even though he is not a Goatwhirler fan himself.

We might, first of all, hit on the idea of using a simple table like this:

Name	Title 1	Title 2	Title3

but, of course, the disadvantage with that is that the number of books which can be recorded for each author is fixed, and some authors may only have one book in the library, in which case a lot of memory is being wasted, while others may have more books than we can allocate space for.

An organization in which each author has his own linked list of book titles gets over this problem. There'll have to be an author array with a corresponding pointer array to point to the first entries in the book lists like this:



To list all the books by a particular author, all we need to do is match the author's name with the corresponding first pointer:

```
500  PRINT "ENTER AUTHOR"
510  INPUT A$
520  FOR P = 1 TO 200              [if there are 200 authors]
530  IF A$ = AUTHR$(P) THEN 570
540  NEXT P
550  PRINT "THERE IS NO REFERENCE TO THIS AUTHOR"
560  RETURN
```

When we get to line 570, P points to the target author, and also to the corresponding position in the FP array. So that FP(P) tells us where to start looking in BOOK$.

So the rest of the routine looks very like the previous linked list print-out program:

```
570  LET P = FP(P)
580  PRINT BOOK$(P)
590  IF PTR(P) = 0 THEN RETURN
600  LET P = PTR(P)
610  GOTO 580
```

All this assumes that the AUTHOR$, FP, BOOK$ and PTR arrays have already been correctly set up. Our librarian will not thank us if he has to know about the internal structure of his data in the primitive way that we've just thought about it. So we need another routine which allows him to enter the data in a more natural way, and sets up the links that we need automatically. I'll leave you to think about that problem. It's quite an interesting little project. Just to give you some pointers (pun!) to it, I'll consider a related routine, one to make an insertion of a new book into an existing author catalogue.

To take a particular example, suppose we add Franz Kafka's *America* into it, keeping a note of where it is (element 8, in our example). At the same time we can set the corresponding element of PTR to zero, since the new book is now the last entry under this author. So PTR(8) = Ø in this case. The only other thing to do is replace the old zero delimiter by 8, so that "AMERICA" will be printed after "THE CASTLE" in the print routine. Of course, it's no good just searching the PTR array for a zero, because we need the one which terminates Kafka's novels. So we have to look for KAFKA F. in AUTHOR$, find 5 in the corresponding FP element, look in PTR(5), find 6, look in PTR(6), find zero and replace it by 8.

The routine is:

```
1500  PRINT "ENTER AUTHOR"
1510  INPUT A$
1520  PRINT "ENTER NEW TITLE"
1530  INPUT NT$
1540  FOR P = 1 TO 1000                    [if BOOKS$ is 1000 elements long]
1550  IF BOOK$(P) = "" THEN GOTO 1570      search for
1560  NEXT P                               1st null entry
1570  LET BOOK$(P) = NT$                   insert new
1580  LET PTR(P) = Ø                       title
1590  For 1 = 1 TO 200                     search for
1600  IF A$ = AUTHR$(I) THEN GOTO 1620     author
1610  NEXT I
1620  IF PTR(I) = Ø THEN GOTO 1650         search for
1630  LET I = PTR(I)                       terminating
1640  GOTO 1620                            zero
1650  LET PTR(I) = P                       replace it with P
1660  RETURN
```

MENUS

We've got the basis, here, of a genuinely useful data-retrieval system. Obviously, there are a lot more routines which would be necessary (for instance we don't have a way of deleting an entry at the moment), and we need a way of linking them together. A convenient technique is to use a menu. When the program is run, it first displays on the screen a list of options which it can execute. So, in this example, we might get something like:

```
LIBRARY RETRIEVAL SYSTEM
------------------------
OPTIONS ARE:
  1. SET UP NEW LIBRARY
  2. INSERT
  3. DELETE
  4. SEARCH
ENTER OPTION (1–4):
```

When you enter one of the options you may get a sub-menu. In this case if 2 is entered, you could get:

```
INSERT ROUTINE
--------------
OPTIONS ARE:
  1. AUTHOR INSERT
  2. NEW TITLE
ENTER OPTION (1/2):
```

This way, all the routines can be written as independent subroutines, the menus can just be printed out at the beginning, and calling the right subroutine can be achieved with:

```
45  INPUT "ENTER OPTION (1 – 4)"; L
50  ON L GOSUB 1000, 2000, 3000, 4000
```

where OPT is the value input at the bottom of the menu. We just have to ensure that the SET UP NEW LIBRARY routine is at 1000, INSERT is at 2000, DELETE is at 3000 and so on. The INSERT routine would have something similar:

```
2015  INPUT "ENTER OPTION (1/2)"; L
2020  ON L GOSUB 2300, 2600
```

The AUTHOR INSERT routine would have to be at 23ØØ and the NEW TITLE routine at 26ØØ.

There's more about breaking programs down into manageable chunks in the section on *Structured Programming*.

There are plenty of other features which you can add to this embryo data retrieval system. How about a subject-index, for example? A SUBJECT$ array will be necessary (with its associated FP array), behaving just like the AUTHOR$ array. There's no point in repeating the information in BOOK$, but the linking pointers will be different, so we need a new PTR array.

At the moment, it is not easy to find out who wrote a particular book. Why not have a set of pointers pointing from BOOK$ and AUTHOR$ to handle that?

And so on, and so on. But I'll hand over to you, at this point. There's a lot of interesting programming here, and a really useful program at the end of it.

Project

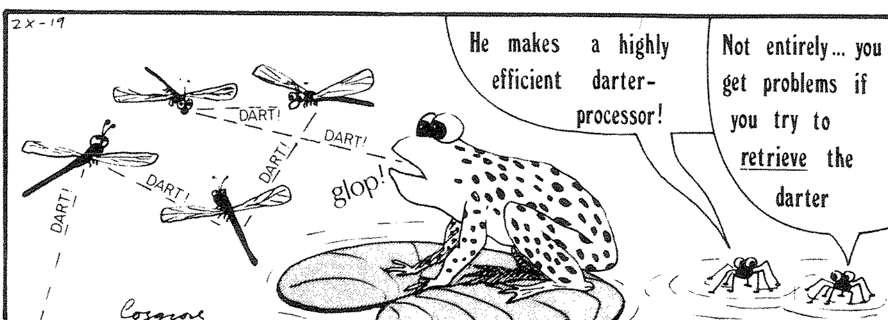
I've alluded several times in this section to problems in Stock Control. It isn't particularly difficult to write a suite of programs to handle the stock for a small business.

You'd start by considering what kinds of data are required, and a suitable structure for these. You'd certainly need things like part number, part description, unit cost, number in stock, location (bin number), reorder level, reorder number, supplier's address. The chances are that arrays organized as tables will do for this, unless there are alternative suppliers of some items, in which case linked lists would be convenient.

Then you'd need routines like these:

1. Add new stock.
2. Remove stock.
3. Interrogate system for (a) number in stock of a given item
   (b) location of a given item
4. Add new item.
5. Delete item.
6. Change supplier of item.
7. Change price of item.
8. Generate orders for items at below reorder level.
9. Generate financial reports (such as average cash tied up in stock).

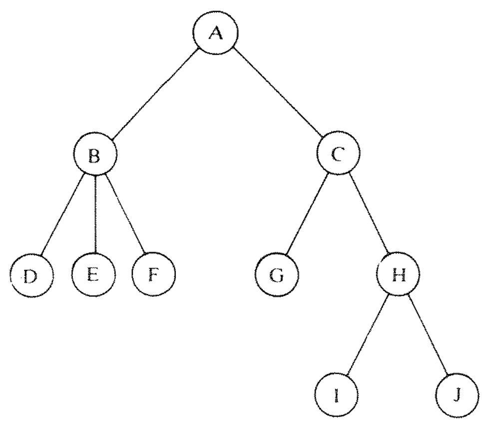
Like all "real" projects this can grow like Topsy. The important thing is to make sure that each routine is independent of the others so that new ones can be added, and old ones can be edited, with the minimum of bother.



An old genealogist's trick, or tasks which branch at every stage.

6 Trees

A *tree* is a structure consisting of *nodes* and *branches*. Each node (except one) has exactly one branch entering it, and may have any number, including zero, leaving it. The exception is the root, which has no branches entering it. A node which has no branches leaving it is called a *leaf*. So, a tree might look like:



The lettered circles are all nodes, joined by straight line branches. "A" is the root and "D", "E", "F", "G", "I" and "J" are all leaves. Implementing this structure isn't much different from implementing a linked list, except that now there is a variable number of pointers from each node. In this case there are never more than three, so a three-column array will do:

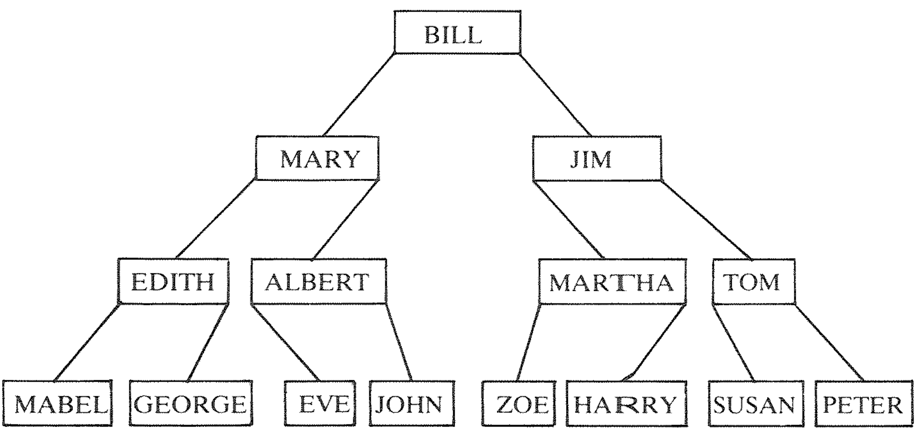
	DTA$	PTR		
1	A	2	3	Ø
2	B	4	5	6
3	C	7	8	Ø
4	D	Ø	Ø	Ø
5	E	Ø	Ø	Ø
6	F	Ø	Ø	Ø
7	G	Ø	Ø	Ø
8	H	9	1Ø	Ø
9	I	Ø	Ø	Ø
1Ø	J	Ø	Ø	Ø

A FAMILY TREE

There are some very straightforward applications for trees, and some less obvious ones.

Let's look at a straightforward one first – the representation of a family tree.

This is another data retrieval problem, really. We store the family tree, and then want answers to questions like "who was X's maternal grandfather?" So we might consider an organization like:



In other words, Jim and Mary are Bill's parents, Albert and Edith are Mary's and so on. Of course, it's an incomplete representation, because there is, for example, no way of telling whether Albert and Edith had more than one child. It's really just that part of the complete tree which affects Bill directly. To get more detail, we would need pointers from

each node to other trees showing the offspring of that node's brothers and sisters; which would lead to other trees in the same way. Boggle, boggle! Let's keep it simple.

The internal structure looks like:

	DTA$	PTR	
1	BILL	2	3
2	MARY	4	5
3	JIM	6	7
4	EDITH	8	9
5	ALBERT	1Ø	11
6	MARTHA	12	13
7	TOM	14	15
8	MABEL	Ø	Ø
9	GEORGE	Ø	Ø
1Ø	EVE	Ø	Ø
11	JOHN	Ø	Ø
12	ZOE	Ø	Ø
13	HARRY	Ø	Ø
14	SUSAN	Ø	Ø
15	PETER	Ø	Ø

SETTING UP A TREE

Genealogists are, in my experience, no more sympathetic to the problems of computer programmers than librarians are, so we'll need a way to set up this structure. Suppose we ask the user to enter the name of a member of the tree, together with his (or her) mother and father, in that order. We don't ask any more than this; that is, we don't insist that BILL is the first entry, for instance.

So we have:

```
1ØØ  INPUT "ENTER NAME, TYPE * TO EXIT"; N$(1)
11Ø  INPUT "ENTER MOTHER"; N$(2)
12Ø  INPUT "ENTER FATHER"; N$(3)
```

Now we insert N$(1), N$(2) and N$(3) into the DTA$ array. Anywhere

will do, because the pointers are going to take care of the links, so the simplest thing is just to load them into the first three available spaces. Rather than have to read DTA$ every time this is done to find some free space, we could keep a pointer to the first free element. This will be 1 to begin with so:

```
90  LET PFF = 1
```

Hang on though! What if one (or even all) of these names have been entered before? We don't want double entries, so we'll have to search the array for each name in turn:

```
160  FOR I = 1 TO 3
170  LET P(I) = Ø
180  FOR R = 1 TO 15
190  IF DTA$(R) = N$(I) THEN P(I) = R
200  NEXT R
210  NEXT I
```

It may have been worrying you why I bothered to enter the names into a new array (N$). Can you see that it's saved repeating lines 17Ø to 2ØØ for three sets of values?

What's the inside loop doing? Well, if the name isn't already in DTA$, P(I) is left at zero. Otherwise, P(I) contains the row where the name is to be found.

Let's follow this through so far with the example. We enter BILL, MARY, JIM. The program searches for these names and doesn't find them so we're left with:

N$

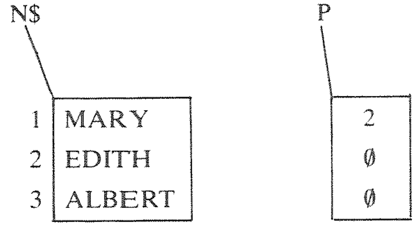
1	BILL
2	MARY
3	JIM

P

Ø
Ø
Ø

We can use this to make the rule: "If an element of P contains zero, the corresponding element of N$ can be entered into DTA$, because it hasn't occurred before."

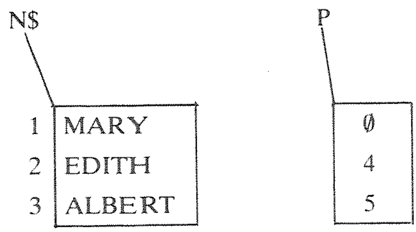
Now we enter MARY, EDITH, ALBERT. This time N$ and P appear as:



So only EDITH and ALBERT need to be copied. As we copy them, let's keep a record of where we put them by holding the row values in the P array:

```
220  FOR I = 1 TO 3
230  IF P(I) < > Ø THEN 27Ø
240  LET DTA$(PFF) = N$(I)
25Ø  LET P(I) = PFF
26Ø  LET PFF = PFF+1
27Ø  NEXT I
```

So now we've got:



and that tells us that the pointers in row 2 should be 4 and 5, or in general, row P(1) contains the pointers P(2) and P(3)!

All we need to write is:

```
28Ø  LET PTR(P(1),1) = P(2)
29Ø  LET PTR(P(1),2) = P(3)
```

That's broken the back of the problem; now for some tidying up. First we need to loop what we've got, to allow all the names to be entered:

```
3ØØ  GOTO 1ØØ
31Ø  END
```

And that means we need a way out of the loop:

```
1Ø5  IF N$(1) = "*" THEN 31Ø      [or GOTO wherever]
```

Finally, we've got to dimension all the arrays:

```
10  DIM DTA$(15)    [if no name exceeds 10 letters]
20  DIM PTR(15,2)
30  DIM N$(3)
40  DIM P(3)
```

One more thing you should notice: since leaves don't have pointers leaving them, the initial values at leaves in the PTR array are never altered. Since these will be set to zero by BASIC when the array is dimensioned, and we're using zero to indicate "no pointer," this is fine unless, in the middle of the program, you want to grow a new tree in the same array. Then, you would have to zero the PTR array to leave it in the state the "Grow a tree" routine expects to find it. It would be safest to do this anyway at the beginning of the routine.

Now we come to a very interesting feature of the program we've just written. We have assumed throughout that the data are entered in the logical order—first BILL, MARY and JIM, then EDITH and ALBERT followed by JIM, MARTHA and TOM etc. But actually the order in which the sets of three names are entered doesn't matter a bit. Try it. Key in the program, tack a routine on the end to point to the DTA$ and PTR arrays and then enter, for instance, ALBERT, EVE, JOHN, then JIM, MARTHA, TOM, then MARTHA, ZOE, HARRY and so on. Of course, the positions of the names in the DTA$ array will vary depending on your order of entry, but the pointers will also change so that they point to the right names.

This is nice, because it allows the user to dredge bits of information out of his memory in no special order (which is how most of our memories work) and enter them while he remembers them. He doesn't have to form a notion of the tree that we've been working from.

Searching the tree for details of a person's parentage is easy. Find the required names in DTA$. The pointer in column 1 of PTR of that row points to his (or her) mother and that in column 2 points to the father. The process can be repeated to look for grandparents, great grandparents and so on. I'll leave the actual coding to you. If you want to accommodate longer names try converting DTA$ to a two-dimensional array and using DIM accordingly.

A common use for trees: storing moves in a game.
Backtrack through the tree to find a winning strategy.
Here the computer teaches itself to win at NIM.

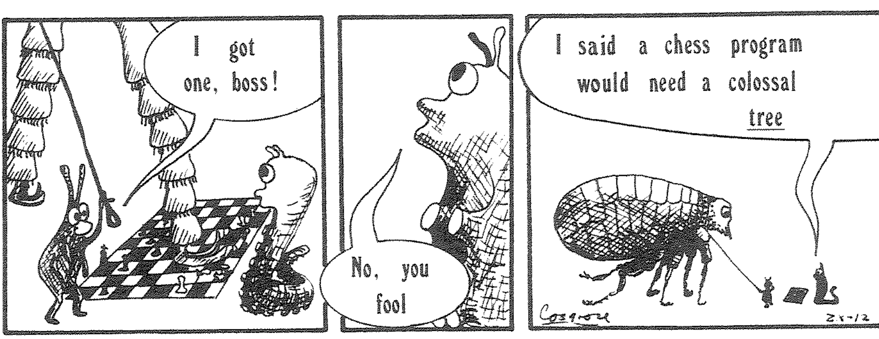
7 Game Trees

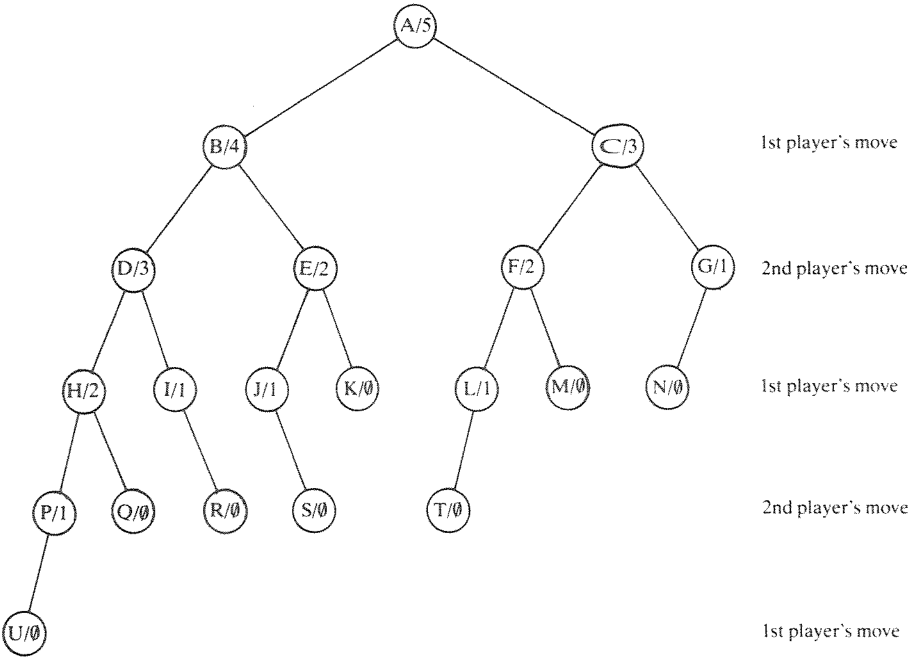
It's possible to represent the moves in a two-person game in a tree. Let's look at a simple example. I'll define the rules of a game as follows.

1. There are two players who take it in turns to move.
2. The initial state of the game is that five matchsticks lie on a table.
3. A legal move consists of removing either one or two matchsticks.
4. A player has won if the other player removes the last match.

You've probably recognized this as a simplified form of Nim (in which there are more matchsticks and more options) and if you've ever played Nim you'll know it's a very simple game, so what we've got is very simple indeed!

The tree shows every position that is possible in the game, and links are shown between successive possible positions. The letters in the nodes are just references, but the numbers indicate the number of matchsticks left at that stage:





Notice that I haven't said anything yet about whether a move is good or bad. For instance, if the game reaches node E, player 1 certainly isn't going to remove two matches and so reach node K because in doing so he loses! But it is a legal move and the tree is only concerned with all possible legal moves.

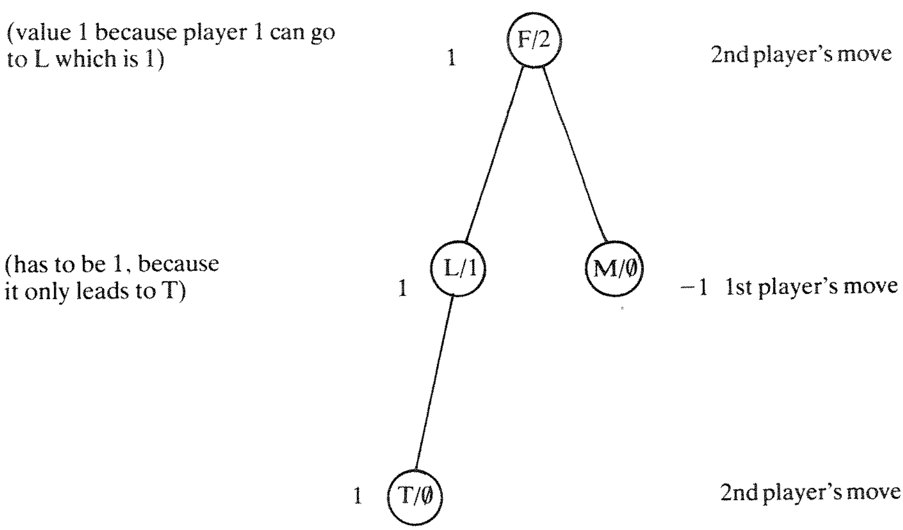
WHEN IS A MOVE GOOD?

Now let's assign some values to the nodes which indicate the quality of a particular move. Obviously, this is easiest at the leaves because we know who's won. Let's define a value 1 to mean player 1 wins and a value -1 to mean player 2 wins. (I've used -1 out of a feeling of symmetry: you *could* try Ø.)

For instance, node U has the value -1 because player 1 picks up the last match. Similarly Q, R, S and T are all set to 1, and K, M and N to -1. In some cases, it's possible to assign values to nodes which aren't leaves. For example, since node P only branches to node U it must have

the same value as node U (i.e. $+1$). This is another way of saying that once the game has reached node P, player 2 is guaranteed to win.

Now we'll make the assumption that each player plays his best move at each stage. That means that if player 1 has the option of going to a node whose value is 1 he'll take it, and so the node from which he's moving can be said to have the value 1. Similarly player 2 will try to make a node $+1$. Using this rule, we can "back up" the tree from the leaves evaluating each node as we go. Look at a small portion of the tree:



If you back up this tree to the root, you'll find the root evaluates to 1. In other words the first player can always win!

What this shows is that Nim isn't a very interesting game. Indeed, no game of which the complete set of moves can be written down is very interesting, because the implication is that it can be played in a totally automatic way, without any guesswork or inspiration. On the other hand, it's exactly that feature which should make it easy for a computer to play!

A PROGRAM FOR NIM

So, how would we go about designing a Nim playing program? First, it's obvious that we need to store the tree:

		NM	NV	PTR	
1	A	5	1	2	3
2	B	4	1	4	5
3	C	3	−1	6	7
4	D	3	1	8	9
5	E	2	1	10	11
6	F	2	1	12	13
7	G	1	−1	14	0
8	H	2	−1	15	16
9	I	1	1	17	0
10	J	1	1	18	0
11	K	0	−1	0	0
12	L	1	1	19	0
13	M	0	−1	0	0
14	N	0	−1	0	0
15	P	1	−1	20	0
16	Q	0	1	0	0
17	R	0	1	0	0
18	S	0	1	0	0
19	T	0	1	0	0
20	U	0	−1	0	0
node reference		number of matches	node value	left pointer	right pointer

We could set this up with a series of input statements inside a loop, or we could "grow" the tree in a similar way to that in which we grew the family tree. Either way, we'll assume that it's available in the form shown.

From here on, the problem's easy. At each stage, the computer has two possible moves, given by the PTR values in the current row. It simply chooses one which leads to a "1" in the node value array. So to start with, we have to initiate a row pointer to the root of the tree, display the number of matches and inform the human player that it's the computer's go (after all, we want the computer to win, don't we?):

```
1000  LET R = 1
1010  PRINT "THERE ARE ";NM(R); " MATCHES ON THE TABLE."
1020  PRINT "IT'S MY TURN"
```

Notice that I've written NM(R) in line 1010 rather than just "5" because this will enable us to use the same statement next time around.

Now to calculate the computer's move:

```
1030  FOR C = 1 TO 2
1040  IF NV(PTR(R,C)) = 1 THEN 1060
1050  NEXT C
1060  LET R = PTR(R,C)
1070  PRINT "THERE ARE " ;NM(R); " MATCHES ON THE TABLE."
```

The loop (1030-1050) looks for a pointer in the current row which points to an NV value of 1. When it finds one it updates the current row to this value. There's no real need for a loop, since there are only two columns (i.e. two possible moves) to consider. But using one means that this form of routine will work for more complicated games where there are dozens of possible moves and correspondingly many columns. We only have to change the "2" in line 1030 to however many columns there are. Note also that we only need to find the first "1" in NV. After all, any "1" leads to a win.

Now we allow the human player a turn:

```
1080  PRINT "YOUR TURN NOW"
1090  PRINT "HOW MANY MATCHES DO YOU WANT TO PICK UP?"
1100  INPUT HM
1110  IF HM > 0 AND HM < 3 THEN 1140
1120  PRINT "NO CHEATING"
1130  GOTO 1090
1140  LET R = PTR(R,HM)
```

Now to see if we've won yet, and if not, to play again:

```
1150  IF NM(R) > 0 THEN 1010
1160  PRINT "I'VE WON AGAIN!": RETURN
```

If we want to be generous, we could allow the user to go first every other game, by passing control to the "computer move" and "human move" routines in the opposite order. That would mean including lines in the move routine to see if the human player has won, and treating the two "play" routines as separate subroutines rather than in-line code. If there's a variable called G which counts the number of games played,

and the two routines begin at 1020 and 1080, as they do here, the calling program could look something like:

```
500  LET G = 0
510  IF G = 2*(INT(G/2)) THEN GOSUB 1020 ELSE GOSUB 1080
520  IF G = 2*(INT(G/2)) THEN GOSUB 1080 ELSE GOSUB 1020
530  LET G = G + 1
540  GOTO 510
```

(don't forget to terminate the two "play" routines with RETURNs).

There are various other "cosmetic" modifications to be made (at the moment it displays: THERE ARE 1 MATCHES ON THE TABLE, for example), but I'm more concerned with another problem:

MAKING THE COMPUTER DO THE WORK

So far, we've assumed that the arrays have to be set up manually, in one way or another. But do they? After all, we set up the tree, and hence the arrays, from a knowledge of the rules of the game, and nothing else. Couldn't we give the computer the rules and get it to generate the array values in a similar way?

Let's give it a whirl. The NM array shouldn't be difficult; we just start with 5 in row 1 and use the rules to subtract 1 or 2 from this at each stage. The PTR array will have to link these values, and so far we have (without ever actually saying so) adopted the convention that the rule "Remove one match" is handled by the first column (rule 1) and the rule "Remove two matches" is handled by the second (rule 2). So we'll stick to that. Of course, the NV array will have to wait until we've grown the tree, because we need to "back up" to get the node values.

Assuming the arrays are already dimensioned, we can start with:

```
 90  LET R = 1                [first row]
100  LET NM(1) = 5            [set the root]
110  LET CP = 2               [set the "current pointer" to the first
                               free row]
120  LET NM(CP) = NM(R) - 1   [rule 1]
130  LET PTR(R,1) = CP        [link the pointer for rule 1]
140  LET CP = CP + 1          [update 1st free row]
150  LET NM(CP) = NM(R) - 2   [rule 2]
160  LET PTR(R,2) = CP        [link the pointer for rule 2]
170  LET CP = CP + 1          [update 1st free row]
180  LET R = R + 1            [go to next node]
```

Now we would like to go back to line 12Ø, of course, to deal with the next node, but we need to avoid getting into an endless loop. We could cheat by remembering that there are twenty rows in the arrays, and branch out of the loop when we've dealt with all twenty; but bear in mind that we're trying to use this game to develop techniques which will still be applicable in more complex situations, when probably we won't know how many nodes there are in the tree.

Actually, there's an implicit rule we haven't used yet; namely, that you can't pick up a match which isn't there. So when matches are subtracted at line 12Ø and 15Ø, we ought to test to see that NM (CP) does not go negative. If it becomes zero, that implies that we've reached a leaf. Maybe we can use this to get out of the loop. If we insert:

```
115  IF NM(R) - 1 < Ø THEN 18Ø
145  IF NM(R) - 2 < Ø THEN 18Ø
```

the "negative match" problem goes away.

Now let's think about the way the pointers, R and CP, behave. CP jumps off at twice the rate of R to begin with, since it's updated twice every loop to R's one. When the leaves start to be reached, CP slows down and eventually stops at 2Ø, because the tests at lines 115 and 145 cause jumps around the "CP update" lines. All this time, R is plodding along, tortoise-like, at one update per loop. When it catches up with CP, we know there have been no forward entries past this row. So all we need is to test if R = CP yet:

```
19Ø  IF R < CP THEN 115
```

Now we're left with generating the NV array values. Obviously we have to start at the leaves. We can find them easily enough, since the NM value at a leaf is zero. But in order to give the node a value we have to know whose turn it was.

Let's set up another array called WPM (which player to move) whose values will be set to 1 or 2 according to whether player 1 or player 2 is about to move. Here it is, added to the previous table.

node reference		NM number of matches	NV node value	PTR left pointer	PTR right pointer	WPM which player to move
1	A	5	1	2	3	1
2	B	4	1	4	5	2
3	C	3	−1	6	7	2
4	D	3	1	8	9	1
5	E	2	1	10	11	1
6	F	2	1	12	13	1
7	G	1	−1	14	0	1
8	H	2	−1	15	16	2
9	I	1	1	17	0	2
10	J	1	1	18	0	2
11	K	0	−1	0	0	2
12	L	1	1	19	0	2
13	M	0	−1	0	0	2
14	N	0	−1	0	0	2
15	P	1	−1	20	0	1
16	Q	0	1	0	0	1
17	R	0	1	0	0	1
18	S	0	1	0	0	1
19	T	0	1	0	0	1
20	U	0	−1	0	0	2

So in row 1, the WPM value is 1. Now, the rest of the WPM array is a bit puzzling, because there are varying numbers of 1s and 2s in each block. In an ideal world they would simply double each time: 1, 2, 4, 8 and so on, because there are two branches leaving every node. They don't of course, because some branches would lead to illegal moves. So we need a limit on the number of rows we examine with each loop. Initially, the first and last entries in WPM are at row 1 so:

```
210 LET FST = 1
220 LET LST = 1
```

To start with, player 1 is making a move so:

```
230  LET PLAYER = 1
```

Now we want to change the player:

```
240  LET PLAYER = (2 AND (PLAYER = 1)) + (1 AND (PLAYER = 2))
```

That's a sneaky piece of code which will swap PLAYER from 1 to 2 and vice versa. Now, we'll loop from FST to LST counting the entries in WPM as we go:

```
250  LET ENTRIES = 0
260  FOR R = FST TO LST
270  IF PTR(R,1) = 0 THEN 330        [this branch doesn't lead anywhere!]
280  LET WPM(PTR(R,1)) = PLAYER
290  LET ENTRIES = ENTRIES + 1
300  IF PTR(R,2) = 0 THEN 330        [neither does this one]
310  LET WPM(PTR(R,2)) = PLAYER
320  LET ENTRIES = ENTRIES + 1    330  NEXT
```

Now look at the next block:

```
340  LET FST = LST + 1
350  LET LST = FST + ENTRIES - 1
```

and see if we're over the end of the array yet:

```
360  IF FST < CP THEN 240
```

EVALUATING THE MOVES

Now we can set the NV values for the leaves:

```
370  FOR R = 1 TO 20
380  IF NM(R) > 0 THEN 410
390  IF WPM(R) = 1 THEN LET NV(R) = 1        [here only for a leaf]
400  IF WPM(R) = 2 THEN LET NV(R) = -1
410  NEXT R
```

Next we search backwards through the tree, linking the node values. There are three situations to consider:

1. There's only one branch, so we can just pass the value back.
2. There are two branches, and the search finds the left link.
3. There are two branches, and the search finds the right link.

In the last two cases, the code is going to be several lines long, testing which pointer points to the maximum node value, and whether we want to pass back the minimum or maximum value, depending on who is about to play. So we'll put it in a subroutine starting at line 8000. The code is:

```
420  FOR R = 20 TO 2 STEP -1
430  FOR RP = 1 TO 20
440  IF PTR(RP,1) = R AND PTR(R,2) = 0 THEN NV(RP) = NV(R)
450  IF PTR(RP,1) = R AND PTR(RP,2) < > 0 THEN GOSUB 8000
460  IF PTR(RP,2) = R THEN GOSUB 8000
470  NEXT RP
480  NEXT R
```

Notice that, to make the code simple, all twenty rows are always searched in the inside loop. Of course, they don't need to be, because there's only one reference to a given pointer in the PTR array. So the code is inefficient, but tidy.

Now for the subroutine: let's make a table showing all the possible combinations of node values being pointed to, and show also what each player would want as a backed-up node value:

Left pointer node value	Right pointer node value	Player 1	Player 2
−1	−1	−1	−1
−1	1	1	−1
1	−1	1	−1
1	1	1	1

In the first and last cases, there is no choice in the backed up value, since both branches lead to the same number.

What we ought to do is determine in each case what the maximum and minimum value is, but this seems a bit like overkill for just two branches. (If there were more, we would have to do this.) Suppose we add the two node values in each line:

−2
0
0
2

Now, only when this value is −2 is a "−1" forced to back up and only when it's 2 is a "1" forced to back up:

```
8000  LET TV = NV(PTR(RP,1))+NV(PTR(RP,2))     [add the node values]
8010  IF WMP(RP) = 2 THEN 8050
8020  LET NV(RP) = 1                      [for player 1 assume node is 1]
8030  IF TV < 0 THEN LET NV(RP) = -1         [if it can't be, set it to -1]
8040  RETURN
8050  LET NV(RP) = -1                    [for player 2, assume node is -1]
8060  IF TV > 0 THEN LET NV(RP) = 1            [if it can't be, set it to 1]
8070  RETURN
```

OTHER GAMES?

And that's it! A computer program which works out its own strategy! OK, the code is less than elegant in places; and here and there I've made use of features of this particular game, rather than making things totally general; but I was concerned with making it as easy as possible to follow the code. (Anyway that's my story.) It shouldn't be difficult to apply these principles to the design of a tic-tac-toe playing program, for instance. (Remember, though, that in this case it's possible for the game to be drawn so node values of zero will appear.)

More complicated games have a serious problem, however. The trees get colossal. Take chess for example. The first player has a choice of 20 moves to start with. (Two moves with each of eight pawns and two moves with each knight). The second player has exactly the same options from each of the 20 nodes just generated, so there are 421 nodes (including the root) after just one move each!

The solution is not to grow the whole tree—it simply wouldn't be possible, however large a machine you had—but to restrict it to, say, five moves ahead. Of course, that means it has to be regrown every move, and it also means that there may not be any leaves in the region we're looking at. That's a problem, because our whole technique has depended on setting the leaf values and back up the tree.

What we have to do is to give values to the terminal nodes in the portion of the tree we have stored, according to some set of rules. At the simplest level we score high if we win material and low if we lose pieces. Then we add some "fine tuning". For example, deduct points if our king is unprotected, add them on if our queen controls the center files. How many points to allot for a particular feature is pretty much a matter

of intelligent guesswork to start with. Scoring will become modified with experience of how well, or badly, the program plays.

I don't recommend that you leap into action and write a chess-playing program, but there are a number of simpler games which are relatively easy to program and at which your TRS-80 can be made to play a surprisingly mean game. Connect-4 and Othello spring to mind. But BASIC versions are pretty *slow*. Read the *Machine Code* section if you want to speed up the action—and then prepare for a long job of programming!

TREES AND INTELLIGENCE

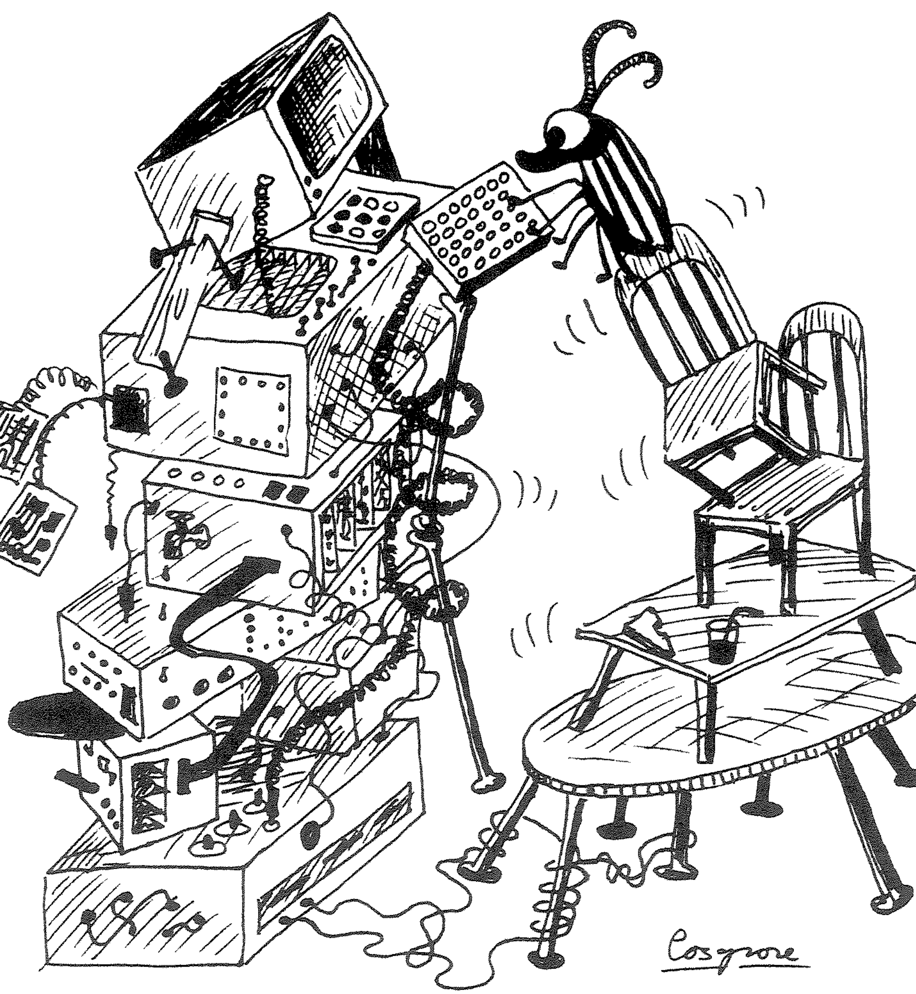
To the outside user, our Nim or Othello player will appear to play pretty sensibly. I will avoid obvious traps, go for winning moves, and in general, will not make aimless plays. That's what a human player would do. So does the program exhibit intelligence?

Well, it depends on what you mean by intelligence. If you mean the ability to model human behaviour in some way (however restricted the field), then the program does do that. If you mean the ability to solve problems which are completely new to the machine, then of course, it can't. But can anybody? Don't we all need some set of rules to work from—some prior knowledge? So maybe the real problem is to devise ways of giving the machine sets of rules (a model of the relevant aspect of the world, if you like) in as convenient a way as possible.

The family tree example is, if you think about it, a simple case of rule giving. After all, a statement like: "Martha and Tom are the parents of Jim" is not, logically, very different from a doctor making a diagnosis saying "A high temperature and spots indicate measles". All we've done is replace "Martha" with "a high temperature", "Tom" with "spots" and "Jim" with "measles". It's possible to design a system which allows a consultant surgeon, for instance, to tell it about symptoms, diagnoses and so on, and which stores all this information in tree-like structures. It can then search the tree to make its own diagnoses when sets of symptoms are given to it. I may even "discover" simplified sets of rules leading to a particular conclusion.

So trees are pretty interesting structures, although handling them can get a little hairy at times. Usually, though, splitting the program down into manageable lumps (as we did in the Nim program), and careful dry running, will allow you to create interesting and useful routines.

Structured Programming



As well as structuring your data, you should also structure the *program*. Essentially this means writing it as a series of linked subroutines—which can be debugged and tested separately, and then strung together, safe in the knowledge that they work. You can name the subroutines in REM statements if you want, to make the listings easier to follow.

More elusive than structure is *style*. Each programmer has his own; and *my* favourite trick may not be to *your* taste. If you compare FRENCH COUNTDOWN and CHECKOUT you'll certainly notice the *difference* in style—but you may not prefer the same style as somebody else. Style is partly clarity and efficiency, and partly good organization; but there's something else, which is hard to pin down.

Rather than lecture to you about structure and style, I've written out three case-histories of the design of longish programs (in BASIC). One is the beginning of a text-editor, one a statistical simulation of supermarket checkouts, and one is an educational program to help teach French (kindly provided by Eric Deeson and adapted to the TRS-80 by Cort Shurtleff). If you don't want to wade through the description, just copy out all the lines in order, as they appear. But I'd rather you took them routine by routine, which is one reason why no complete listing is given on its own.

Beginning steps on a text editor, which give insight into program structure, word processors, and the way the TRS-80 works.

8 Zedtext-80

You hear an awful lot these days about *word processors* or *text editors*: programs that turn a microcomputer into an intelligent typewriter able to edit and manipulate a written text. The TRS-80 facilities for on-screen editing of programs amount to a rudimentary text editor. You can buy very sophisticated word processors that can paginate, justify and index text automatically. Zedtext-80 won't act as a practical text-editor as it stands, however with a little ingenuity you can add a multitude of text-editing functions. If you have a line printer there's no telling what you could do!

CHOOSING A DATA STRUCTURE

A text-editor must accept textual input, and *process* it in various ways. Before starting on the program to do this, we must first work out the most suitable structure for the data—the text itself. But before structuring the data, we have to consider what we intend to do with it. Vicious circle? No, because we can think pretty loosely at first, and write a few experimental programs to test our hunches.

First we must be able to enter text and store it in memory. A second feature is that we should be ooble to delete mistakes like that one, to produce a clean text. We should also be able to append text (or insert it in the middle like this) to existing text, just as this sentence is appended to the previous one. Numerous other features would be desirable—the ability to remember a segment of text and copy it into a chosen place, the ability to save the results on disk or tape, and so on—but the above will be enough for deciding on a data structure.

So let's think about the possibilities.

(a) Think of text as a one dimensional array of single characters:

T	H	I	S	□	I	S	□	A	□	T	E	X	T ...
1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14 ...

(I've drawn it horizontally—the boxes indicate a space, which is a character after all.) The main problem with this is that to insert material in T$ we need to shift lots of characters up several spaces, using code like:

```
1ØØØ  FOR K = TOP TO BOTTOM STEP - 1
1Ø1Ø  LET T$(K) = T$(K - 92)
1Ø2Ø  NEXT K
```

(say). We start at the top and move downwards, of course, to avoid erasing sections of text not yet moved.

This looks as if it would run slowly on a text of say, 3ØØØ characters; otherwise it has the advantage of simplicity.

(b) *Linked Lists.* Arrange the text in a list, with pointers to the next character.

Number	Character	Pointer
1	T	2
2	H	3
3	I	4
4	S	5
5	□	6
6	I	7
...	... ...	...

Modifications of the kind "insert" or "delete" are now very easy. For example, to change the text to

THIS□IS□NOT□A□ TEXT...

all we do is add to the *end* of the list

15	N	16
16	O	17
17	T	18
18	□	19

and modify *two* pointers to get the characters in sequence:

8	□	<u>15</u>
18	□	<u>9</u>

Deletions are equally easy: you just change one pointer to reroute past existing text. One minor problem: the deleted text is still "there" in the sense that it's taking up memory space—it's just that you won't *find* it by tracking through the list following the pointers, because nothing points *into* it. If memory is short, it's a nuisance wasting it in this way. There's a good way out, though: put up some kind of flag in the deleted entries, and every so often do a "garbage collection run" which goes through the list renumbering everything and getting rid of the flagged entries. But such garbage runs will be slow to operate, so the best time to do it is when you get fed up and go off to get some coffee. Moral: don't collect garbage until the memory is close to full.

(c) variant on (b): use linked lists of *words*. Now you'll have a genuine *word*-processor, but it won't be very good at handling symbolic text, e.g. program listings.

The above discussions are very theoretical. In practice, which of these structures you use depends heavily on the particular architecture of the machine you are using. So let's see how the TRS-80 copes.

(c) turns out to be a bit awkward because string arrays always have to have a fixed length of word. This means you have to dimension the array to fit the longest word likely to be used, say

```
10  DIM T$(500,21)
```

for 20 character words. But now our sample text above goes in as:

1	THIS□□□□□□□□□□□□□□□□□□□□	2
2	IS□□□□□□□□□□□□□□□□□□□□□□	3
3	A□□□□□□□□□□□□□□□□□□□□□□□	4
4	TEXT □□□□□□□□□□□□□□□□□□□	5
	....	

which wastes *acres* of memory. Now by being clever, you could doubtless take care of some of this; but it takes time and program space, and it smells complicated.

So ditch that one.

Linked lists look attractive, in fact many commercial editors do use linked lists with the programs written in machine code for extra speed. But on the the TRS-80 there's a simpler way: we could use the video

memory map and treat it as a large string array. We write text on the screen and then use PEEK and POKE to insert it into a block of RAM. This has an advantage of simplicity but when I tried it, it took several minutes to transfer into memory a single screen of text. Ditch that one until you get to machine code—it has special features for the transfer of blocks of code.

We end up using (a) above with a difference. We tie together individual strings in a string array to form a single large string. As it stands it is a bit clumsy. The text is stored in "pages" of 256 characters. We use elements of the array to store the contents of any given "page" and use other string variables to act as a sort of messenger RNA to move blocks of text. Later when you have machine code under control you can use actual blocks of addresses in RAM as memory storage. One advantage to storing text in a string is that we can print onto the screen with the PRINT@ statement.

So after all that, it transpires that there are very good reasons to select a simple *Data Structure*: the text will be stored in a simulated single string of length specified by a DIM statement

As it happens, this has expository advantages too: it's easier to see how the various subroutines work in this rather transparent data structure. In addition, you can later write simple block transfer subroutines in machine code that will easily be patched into your BASIC program. These machine code subroutines will substantially increase the execution speed.

INITIALIZATION

For demonstration purposes we will limit our simulated long string to 12 pages, each containing 255 characters. As you type in a character it is simultaneously printed on the screen and concatenated onto a string array. In this section we need to take care of reservation of memory, initialization of variables, and protecting the top of the screen from scrolling away. Here is the groundwork needed:

```
10  REM *** ZEDTEXT - 80 ***
20  CLS
30  CLEAR 4000
40  PRINT @24, "ZEDTEXT - 80"
50  ZZ = 3 : P = 1 : CP = 1
60  DIM T$(12)
70  POKE 16912, 3 : REM SCROLL PROTECT
80  ES$ = "
                    "
```

```
9Ø  PRINT @128,"
100  GOTO 5ØØ
```

The string ES$ consists of 64 empty spaces, ZZ is the screen-full flag, CP stands for current write page and P indicates the actual page. To understand what these do you need to examine the section of the program that is used to enter text, the Write Loop. Line 9Ø may be a bit hard to read, it prints a solid line across the screen to separate our scroll protect area from the rest of the screen. Our goal is to smoothly enter text on the screen *and* store it the same time. The following program lines do the job.

THE WRITE LOOP

This section of the program has a simple skeleton:

```
5ØØ  REM *** WRITE LOOP ***
51Ø  PRINT @ 64*3, ""
52Ø  A$ = INKEY$ : IF A$ = "" THEN 52Ø

61Ø  PRINT @ CP*256+CI-128,A$ : CI = CI+1

67Ø  GOTO 52Ø
```

The INKEY$ function retrieves a character from the keyboard buffer and is PRINTed @ the appropriate place on the screen. The variable CP has no duty at this point; its presence is needed later to place the text at the right screen location. We will display only three "pages" on the screen at any given time and CP will indicate which page is active at the moment. CI tells us the location of the next character of text. Since a string variable can hold only 255 characters, we need to reset the variable CI back to Ø at the end of the page. The −128 in the arithmetical expression in Line 61Ø slides the entire display up two lines. (Try deleting it to see what happens.)

One advantage of our choice of this design is that we get an uppercase/lowercase toggle for free—pressing the SHIFT and the Ø keys simultaneously will do the trick.

The next set of program lines store the text in an one dimensional string array, and handle the bookkeeping when a page is full:

```
62Ø  T$(P) = T$(P)+A$
63Ø  IF CI = 255 THEN P = P+1 : CI = Ø : CP = CP+1
67Ø  GOTO 52Ø
```

All well and good but what happens when the screen is full? We need to be able to scroll the text up and get a fresh screen to work on. First we test to see if the third page is full:

```
650  IF P = ZZ THEN GOSUB 3000
```

You could just test to see if CP = 4 but using ZZ will free CP if you want to use it in the Edit Mode of the program. Here is the Scroll subroutine that is called for:

```
3000  REM *** MOVE IT UP ***
3010  CLS : PRINT@ 256,T$(P-1)
3020  CI = 0 : ZZ = ZZ + 1 : CP = 2
3030  RETURN
```

This subroutine starts by clearing the screen. (Since we have activated the scroll protect feature, the message on top is preserved.) It then prints the last "page" at the top of the screen. To continue writing, we have to adjust the variables appropriately. A little experimentation yielded the above results.

One final detail, the cursor:

```
660  PRINT@ CP*256+CI-128
```

This will be written over automatically when you type your next character.

A CARRIAGE RETURN

An essential feature of any word processor is a *carriage return*. Obviously computers don't have carriages, what we mean is a way to signal the TRS-80 that we wish to move to the next line. The ENTER key is the key to use. It's ASCII code is 13. First edit Line 600 to match:

```
600  IF S<32 OR S>127 THEN GOTO 2000
```

Now add:

```
2000  REM *** COMMAND DECODE ***
2010  IF S<>13 THEN 2070
2020  PRINT @ 256*CP+CI-128," "
```

This gives a bit of space to work with and also erases the cursor.

Our next problem is to make sure that the text in memory matches the screen after a carriage return. It's easy enough to get the cursor to skip to the next line, the problem is that we have to add all the spaces we have just created to to the text variable; T$(P). That's why I put in

the variable ES$ back in the initialization, it is full of handy empty spaces. Add:

```
2030  Q = INT (CI/64) : W = ((Q+I)*64)-CI
2040  IF Q = 3 THEN W = W-I
2050  A$ = LEFT$ (ES$,W) : CI = CI+W
2060  GOTO 620
2070  GOTO 520 REM TEMPORARY
```

Line 230 calculates the number of spaces remaining. This is a useful math trick so it's worth a little thought. Line 2040 betrays a glitch in our program design. Since a string variable can only store 255 characters every fourth line will be one space short. At this point we will live with it but we do need to add 2040 to make the math come out right. You can patch this up later by inserting some extra lines.

Line 2050 defines A$ to be the correct number of spaces and Line 2050 sends the computation to the right place in the Write Loop.

THE DELETE KEY

Though most editing will take place in edit mode, it would be a convenience to have a delete key in Write Mode. The left-arrow key is the obvious choice. The key has ASCII code 8, so the first step is to set a trap:

```
530  S = ASC(A$)
540  IF S< >8 THEN 600
```

The space between Lines 540 and 600 are reserved to deal with the bookkeeping involved in deleting the last element of the text string T$(P).

```
560  CI = CI-1 : T$(P) = LEFT$(T$(P),CI)
570  PRINT @ ((CP*256)+CI-127), " "
580  GOTO 660
```

If you test the program at this point you will note that it works fine except when CI = 0. (That is when you are at the end of a "page". The following line resets the variables when CI = 0.

```
550  IF CI = 0 THEN CI = 255 : P = P-I : CP = CP-I
```

You now have a text writer that stores the text in a string array. This should give you some insight into what happens when you start writing ambitious programs. There are two other major components to Zedtext-80 that you can tackle if you want: storage on disk (or tape) and Edit mode. My advice is to finish the book and then, if you're still

interested, finish Zedtext-80 with your new found skills. To get you started here's a way to access your screen editor and the Main Cursor loop...

EDIT MODE

The idea behind a screen editor is that you can move the cursor anywhere on the screen and direct the computer to take some appropriate action, say insert or delete text. In addition there should be commands to scroll the screen up and down so that you can inspect any part of your text. If you take a structured approach, you can keep adding more and more features as you figure out how to devise them. If you *don't* take a structured aproach, you'll likely start developing increasingly frustrating cul-de-sacs. Your choice.

This final bit of code allows you to exit Write Mode when you press the clear key. (ASCII code 31) At that point you can move the cursor anywhere on the text screen with suitable combinations of the SHIFT and right and left-arrow keys.

```
2070  IF S = 31 THEN 5000 ELSE 520
5000  REM *** EDIT MODE ***
5010  S1=CI : S2 = CP : S3 = P
5020  CI = 128 : MM = 15360
5030  CZ = PEEK(MM+CI) : POKE MM+CI,95
5040  A$ = INKEYS: IF A$ = "" THEN 5040
5050  S = ASC(A$)
5060  IF S = 9 AND CI<896 THEN POKE MM+CI,CZ : CI = CI+1 : GOTO
       5030
5070  IF S = 8 AND CI>192 THEN POKE MM+CI,CZ : CI = CI-1 : GOTO
       5030
5080  IF S = 24 AND CI > 191 THEN POKE MM+CI,CZ : CI = CI-64 : GOTO
       5030
5090  IF S = 25 AND CI < 831 THEN POKE MM+CI,CZ : CI = CI+64 : GOTO
       5030
7000  GOTO 5020
```

Obviously my choice of line numbers throughout this program was inspired by hindsight. To make room for the ongoing improvements you can simply increment by 100 *or* you can budget your program lines in advance. You might say that structured programming is hindsight applied ahead of time. The next two chapters demonstrate how to take a program goal and work it through to fruition using the structured approach.

A simulation of customer-flow in a supermarket, which can be adapted to the allocation of hospital beds and the queues at filling stations.

9 Checkout

This is an example of a computer simulation. A supermarket manager wants to know how many checkouts to keep open at a given time of day, subject to information on the rate of arrival of customers, distribution of time spent dealing with each customer, and so on. The program will attempt to approximate the true sequence of events, keep an eye on queue lengths, waiting times, number of checkouts idle, and so forth, and present its results. This (ideally) would help the manager decide how many checkouts to keep open, and hence how many checkout staff to employ.

The actual program below is just a first step towards a practical simulation; but by the time we've written it we'll be well aware of what extra features it would need, and how to extend it. Meanwhile we'll keep it relatively simple.

DATA STRUCTURE

What is a good data structure for dealing with N supermarket queues? It's no accident that the word "queue" appears here: the corresponding data structure is designed to do exactly what a real queue does, namely accept input at one end and push things out at the other. What we want is a list of N queues: that is, a *queue array*. Recall that to set up a queue of length 25 (say) we use DIM Q(25), and then special routines to enqueue and dequeue entries. For a queue array we do exactly the same thing, but adding an extra dimension. If CN is the required number of queues (or checkouts) then we'll need DIM Q(CN,25). The "head" and "tail" pointers H and T, and the lap counter L, also become arrays of size CN.

OPERATION

We'll need a main routine to process the queues, say at time steps of 1 minute (simulated time, not real time, that is !). At each time step this will have to:

1. Decide how many new customers join the queues.
2. Decide how long each of them will take to pass through the checkout (i.e. how many goods they've bought).
3. Enqueue them according to some reasonable "customer strategy".
4. Decrease by 1 minute the waiting time of the customer at the head of each queue.
5. Dequeue customers whose time has decreased to Ø.

In addition we'll arrange for:

6. The production of a graphic display of the current situation.

After a chosen number of time steps (say 1ØØ) the program should stop, and display an analysis of the way the queues behaved. Some useful quantities would be:

- The average length of a queue.
- The maximum length of a queue.
- The average time a customer spends waiting in a queue.
- The maximum time a customer spends waiting in a queue.
- The average number of checkouts idle.

These quantities will be evaluated by various "system variables", and the main program must keep these up to date; so we'll also need:

7. Update the system variables.
8. Display analysis at end of simulation run.

ENQUEUE AND DEQUEUE

Let's build these subroutines first, because we've already got them worked out (see page 22). Assume we are working on queue number 1 and wish either to enqueue an extra number V, or dequeue a number and call it V.

Initialize:

```
 5 CLS
1Ø DIM Q(9,25)
2Ø DIM H(9)
```

```
3Ø  DIM T(9)
4Ø  DIM L(9)
```

are envisaging a maximum of nine queues here. (To save memory we could first input the number CN of checkouts, then define CN and change all 9s above to CN; but there's plenty of room in 16K so why bother?)

```
1ØØØ  REM *** ENQUEUE ***
1Ø1Ø  IF H(I) = T(I) AND L(I) = 1 THEN GOTO 1Ø7Ø
1Ø2Ø  LET Q(I,T(I)) = V
1Ø3Ø  LET T(I) = T(I)+1
1Ø4Ø  IF T(I) > 25 THEN LET L(I) = L(I)+1
1Ø5Ø  IF T(I) > 25 THEN LET T(I) = 1
1Ø6Ø  RETURN
1Ø7Ø  PRINT @896, "QUEUE FULL"
1Ø8Ø  STOP
```

(If a queue of length 25 gets full, the run stops; but this indicates that queues have grown much too large for comfort, and more checkouts are needed.)

```
11ØØ  REM *** DEQUEUE ***
111Ø  IF H(I) = T(I) AND L(I) = Ø THEN RETURN
112Ø  LET V = Q(I,H(I))
113Ø  LET H(I) = H(I)=1
114Ø  IF H(I) > 25 THEN LET L(I) = L(I)-1
115Ø  IF H(I) > 25 THEN LET H(I) = 1
116Ø  RETURN
```

As it happens, when we dequeue, V is always Ø; but we don't know this at this stage in the writing; anyway, it would be nice to check. We're not worried if a queue gets empty (unlike ENQUEUE where a full queue spells disaster) so line 111Ø doesn't ask for a "QUEUE EMPTY" display as line 1Ø7Ø did.

The initialization of T and H isn't quite what we want, though: the ENQUEUE and DEQUEUE routines assume H and T start at 1, not Ø. So we need to add:

```
5Ø  FOR I = 1 TO 9
6Ø  LET H(I) = 1
7Ø  LET T(I) = 1
8Ø  NEXT I
```

GRAPHICS

The next routine sets up CN "checkouts" with spare lines for the queues to be printed out (by another subroutine which follows).

```
1200  REM *** CHECKOUT GRAPHICS ***
1210  CLS
1220  FOR I = 1 TO CN
1230  PRINT @74+64*I,CHR$(141);CHR$(140);CHR$(140);CHR$(48+I);
      CHR$(140)
1240  NEXT I
1250  RETURN
```

To print the queues:

```
1300  REM *** QUEUE PRINT ***
1310  LET X = H(I)
1320  LET Q$ = ""
1330  IF Q(I,X) = Ø THEN GOTO 135Ø
1340  LET Q$ = Q$+CHR$(48+Q)              build inverse video
1350  IF X = T(I) THEN GOTO 139Ø           copy of queue
1360  LET X = X+1
1370  IF X > 25 THEN LET X = 1
1380  GOTO 133Ø
1390  PRINT @79+64*I, Q$; " ";             print it
1400  RETURN
```

Each queue is stored in a string variable, the numbers in the queue represent the waiting time for that individual (i.e. the time needed for him/her to pass through the checkout).

CUSTOMER HANDLING

Next, the subroutines dealing with customers. First we need to decide how many new customers join the queues during one time-step (one "minute"). For simplicity I'll use a random number of arrivals, generated in the obvious way, to produce an *average* arrival rate AR. I'll input AR itself later.

```
1500  REM *** ARRIVALS ***
1510  LET NA = INT(2*AR*RND(Ø))
1520  RETURN
```

Well, that's almost too trivial to have as a subroutine at all; but I'm going to leave it like that in case someone wants a more complicated distribution of arrivals at some later stage in the development of the program. It will be easier to change it if it's a subroutine on its own. NA, of course, is the number of customers arriving.

What do the customers do when they join a queue? They sit and wait. The question is, for how long? Each customer will have bought a certain number of goods, which will take a certain time to process. For ease of display and computation, I'm going to assume that each customer is assigned a waiting (i.e. processing) time that is a whole number of minutes, and at least 1.

For illustrative purposes, suppose we want to produce a distribution of waiting times, so that of every 3Ø customers, on average:

```
 4 customers take 1 minute  to process
 6    ''        '' 2 minutes ''    ''
1Ø    ''        '' 3   ''    ''    ''
 8    ''        '' 4   ''    ''    ''
 2    ''        '' 5   ''    ''    ''
```

Looks like a messy thing to program, but it's not. Take an easier case: suppose that out of every 6 customers, on average, we want

```
1 customer  to take 1 minute
1    ''         '' 2 minutes
  ..........
1    ''         '' 6 minutes
```

Obviously we "roll a die" to decide the length of time for each customer. That is, we'd use INT (1 + 6 * RND). For the more complicated distribution above, we need to concoct a 3Ø-sided die, such that 4 sides are labelled "1", 6 sides "2", 1Ø sides "3"; 8 sides "4", and 2 sides "5".

Here is such a die:

```
LET D$ = "111122222233333333334444444455"
```

If we can pick a digit out of D$ at random, we can "roll" the die too; and that's not hard to arrange:

```
LET XX = INT(1+3Ø*RND(Ø))
LET NT = VAL(MID$(D$,XX,1))
```

OK, here's the scenario. When a new customer arrives, (s)he is going to select which checkout to go to by some *strategy*. The waiting time

is assigned using the "die" D$. For the moment I'm only going to illustrate the simplest strategy, "choose a queue at random"; but you might like to think about modifying the program to allow two others:

1. Choose the shortest queue (or at random from the shortest ones if there is more than one)—I'll add this in later.
2. Choose the queue with shortest total waiting time (which a customer could estimate fairly well by looking at how much stuff is in people's baskets, so it's reasonably realistic).

That gives us two new subroutines:

```
1600 REM *** STRATEGY ***
1610 LET I = 1 + INT(CN*RND(0))
1620 RETURN
```

(which decides which checkout a given customer will go to, and calls it I):

```
1700 REM *** NEXT WAITING TIME ***
1710 LET XX = INT(1+30*RND(0))
1720 LET NT = VAL(MID$(D$,XX,1))
1730 RETURN
```

We've also got to initialize D$:

```
90 LET D$ = "111122222233333333334444444455"
```

MAIN PROGRAM

Now comes the bit that organizes all the rest: it works out what happens during the Nth time-step. First we initialize a variable to count the steps,

```
100 LET STAGE = 0
```

Next we arrange for the operator to input a couple of variables that need to be decided before the subroutines can work:

```
300 PRINT "SUPERMARKET CHECKOUT SIMULATION"
310 INPUT "ENTER NUMBER OF CHECKOUTS ( 1 - 9 )"; CN
320 PRINT "ENTER CUSTOMER ARRIVAL RATE"
330 INPUT "( AVERAGE PER MINUTE )"; AR
400 GOSUB 1200
```

Now we're all set for the action:

```
500  REM *** ONE TIME STEP ***
510  LET STAGE = STAGE+1
520  PRINT @965, "STAGE = ";STAGE;
530  GOSUB 1500                              ]- number arriving?
535  IF NA = 0 THEN GOTO 590
540  FOR X = 1 TO NA
545  GOSUB 1600                              ]- choose queue
550  GOSUB 1700                              ]- find waiting time
560  LET V = NT
570  GOSUB 1000                              ]- enqueue
580  NEXT X
590  FOR I = 1 TO CN
600  IF H(I) = T(I) AND L(I) = 0 THEN GOTO 640
610  IF Q(I,H(I)) = 0 THEN GOSUB 1100         decrement front
620  IF Q(I,H(I)) = 0 THEN GOTO 640           of queue, and
630  LET Q(I,H(I)) = Q(I,H(I))-1              dequeue if zero
640  NEXT I
650  FOR I = 1 TO CN
660  GOSUB 1300
670  NEXT I                                  ]- print queues
680  GOTO 500
```

Not bad. But it won't ever stop! If we want to run a specified number of stages, then we'll need to tell the machine how many, and when to come out of the loop from 680 back to 500. So we'll add:

```
360  INPUT "ENTER NUMBER OF STAGES IN RUN"; NSTAGE     check if
675  IF STAGE = NSTAGE THEN GOTO 2000                 run has
                                                      ended
```

At line 2000 we'll put a routine that produces an analysis of the main features of the run.

That raises all sorts of interesting questions, because we'll have to keep track of those features as the run proceeds, and so far we haven't built anything into the program that will do this. So let's take a look at the problems involved.

SYSTEM VARIABLES

To keep track of what's happening, we need to set up various new variables, update them as necessary, and use them to calculate the in-

formation required in the final analysis of the run.

First let's decide what we'd like to know. A few suggestions:

ML = maximum length of a queue.
AL = average length of a queue.
MW = maximum time a customer waits in a queue.
AW = average time a customer waits in a queue.
AE = average number of empty queues (i.e. idle checkouts).

Of these, ML and MW are the easiest to deal with. All we have to do is scan the queue lengths (wait times) at each stage (as each customer joins) and increase ML or MW if this number is larger than the current value. So first we initialize:

```
110 LET ML = 0
120 LET MW = 0
```

Then we need two arrays W and N which hold the total waiting time and length of a given queue:

```
130 DIM W(9)
140 DIM N(9)
```

Now for those averages. You have to think quite carefully what they mean. The way to get an average is to total a lot of numbers and divide out by how many of them there are—but you have to total the right things! The sensible way is to keep a running total, and a running count, and divide one by the other.

Thus to get the average length of the queue, we need a running total length TL, which is updated for each queue once per stage. The running total of queues involved, at CN per stage, is just CN * STAGE. So AL = TL/(CN * STAGE). But where do we update TL?

First we initialize it:

```
150 LET TL = 0
```

Then we decide where to find the queue lengths: the easiest place is during the QUEUE PRINT routine:

```
1395 LET N(I) = LEN Q$
```

We haven't finished with TL yet, but let's put the rest of the calculation into an UPDATE subroutine, to be written below; and start working on AW, the average waiting time. Averaged over what? Average *per customer*. A given customer's waiting time is the total waiting time of

the queue he joins, at the moment he joins it, *plus* his own waiting time. So most of the updating will get done during the ENQUEUE subroutine, where the customer joins the queue. Further, we need to keep a check on the total number of customers that has passed into the system, with a variable TCUS (for Total CUStomers). And TW will total the waiting times. Sounds messy? Just keep a clear head!

```
160  LET TW = Ø
170  LET TCUS = Ø
```

Updating is done here:

```
642  FOR I = 1 TO CN
644  LET W(I) = W(I)-1+(W(I) = Ø)
646  NEXT I

1ØØ2  LET TCUS = TCUS+1
1ØØ4  LET W(I) = W(I)+V
1ØØ6  LET TW = TW+W(I)-1
```

UPDATING

The update routine also takes care of the "average number of checkouts empty" variable, by making a running total TE and dividing by the value of STAGE. Initialize:

```
18Ø  LET TE = Ø

18ØØ  REM *** UPDATE ***
181Ø  LET EC = Ø                                  set empty counter EC
182Ø  FOR I = 1 TO CN
183Ø  IF N(I) > ML THEN LET ML = N(I)             update ML
184Ø  IF W(I) > MW THEN LET MW = W(I)             update MW
185Ø  IF N(I) = Ø THEN LET EC = EC+1              update EC
186Ø  LET TL = TL=N(I)                            update TL
187Ø  NEXT I
188Ø  LET TE = TE+EC                              update TE
189Ø  LET AL = TL/(CN*STAGE)
19ØØ  IF TCUS = Ø THEN GOTO 192Ø                  compute averages
191Ø  LET AW = TW/TCUS
192Ø  LET AE = TE/STAGE
193Ø  RETURN
```

Before we get too carried away with the amazing success of this subroutine, we'd better take care of one minor point: there's no way to get into it yet. We need to modify the main program:

```
672  GOSUB 18ØØ
```

ANALYSIS

Coming down the home stretch now. All we need is to work out a few more system variables, and print it all out.

```
2ØØØ  REM *** ANALYSIS ***
2Ø1Ø  CLS
2Ø2Ø  PRINT "      ANALYSIS",,,
2Ø3Ø  PRINT "NUMBER OF CHECKOUTS=  ";CN
2Ø4Ø  PRINT "CUSTOMER ARRIVAL RATE=   ";AR
2Ø5Ø  IF SNUM  = 1 THEN PRINT "CUSTOMER STRATEGY:RANDOM"
2Ø55  IF SNUM  = 2 THEN PRINT "CUSTOMER STRATEGY :'T LINE"
      SHORTEST  LINE"
2Ø6Ø  PRINT "NUMBER OF STAGES=  ";NSTAGE
2Ø7Ø  PRINT
2Ø8Ø  PRINT "MAXIMUM LENGTH=  ";ML
2Ø9Ø  PRINT "AVERAGE LENGTH=  ";AL
21ØØ  PRINT "MAXIMUM WAIT     =  ";MW
211Ø  IF TCUS = Ø THEN GOTO 215Ø
212Ø  PRINT "AVERAGE WAIT     =  ";AW
213Ø  PRINT "AVERAGE EMPTY  =  ";AE
214Ø  STOP
215Ø  PRINT "AVERAGE WAIT     =Ø"
216Ø  GOTO 213Ø
```

A SAMPLE RUN

Once you've got the above listings typed in (and no doubt re-debugged them, typing errors being almost unavoidable in this game) the simulation is ready. Type RUN. When asked, INPUT the number of checkouts (say 7), the arrival rate (say 4) and the length of run (say 1Ø, since the program runs fairly slowly). You'll get a graphic display, changing at each stage according to the way the customers behave. Then, after the 1Øth stage, comes the Analysis.

Here's a typical series of runs, with the arrival rate AR = 3 throughout, and NSTAGE = 2Ø; all the values CN = 1, 2, 3,....,9 have been tried in turn. (You can of course get the computer to do all this for you, by suitable modifications—but for an exploratory attempt it's not really worth the bother.) The analysis is summarized in a table:

CN	ML	AL	MW	AW	AE
1	QUEUE FULL AT STAGE 15—RUN STOPPED				
2	23	12.675	62	31.711864	Ø.2
3	15	5.8333333	46	18.44	Ø.2
4	10	3.9125	27	12.5	Ø.25
5	11	3.31	31	11.565217	Ø.9
6	7	1.7	18	7.44186Ø5	2
7	6	1.81642857	15	6.7884615	1.75
8	5	1.Ø3125	16	5.4324324	3.85
9	4	1.Ø722222	13	4.4444444	3.1

So, for example, with 7 checkouts the longest queue in the 2Ø stages run was 6; the average was about 1.8, the longest wait was 15 minutes, the average wait about 6.8 minutes, and on average 1.75 checkouts were idle. The extra decimal places provide a totally spurious suggestion of accuracy, and shouldn't be believed.

These are probably barely acceptable figures: you wouldn't want customers to wait much longer than that! The figures with 6 checkouts are a trifle worse; with 8, better on average; with 9 better still. With 5, they're terrible! So you conclude that you need about 7 checkouts.

Of course, 2Ø is a rather short run; but armed with this rough estimate, you can try longer runs with 7 or 8 checkouts. I took NSTAGE = 1ØØ, AR = 3, and got:

CN	ML	AL	MW	AW	AE
7	11	3.5857143	34	11.7Ø1961	Ø.95
8	12	1.875	38	7.962Ø253	2.85

Now 7 looks less good; and 8 is acceptable. Conclusion: we need 8 checkouts.

ALTERNATIVE STRATEGIES

You won't always get exactly these figures, of course, because of the random element in the simulation; but they're fairly typical. However, you may well suspect that the figures are unreasonably distorted by the choice of customer strategy: in practice, shoppers do *not* choose the checkout to join at random!

We can easily modify the existing program to permit different strategies: simply change the subroutine at 16ØØ, or better still, allow a range of strategy subroutines and call the required one when starting the run. Note that it is this easy precisely *because* we have broken the program up into subroutines.

For example, suppose the customer sizes up the queues, sees which are shortest, and joins one of the shortest ones (at random if there are two or more). Then we would need a new strategy subroutine; and for ease of modification we'll put it at line 3ØØØ. Here's the guts of it:

```
3ØØØ  LET Z = 1
3Ø1Ø  LET MQL = 25                          find length
3Ø2Ø  FOR I = 1 TO CN                       of shortest
3Ø3Ø  IF N(I) < MQL THEN LET MQL = N(I)     queues
3Ø4Ø  NEXT I
3Ø5Ø  FOR I = 1 TO CN
3Ø6Ø  IF N(I) < > MQL THEN GOTO 3Ø9Ø        list shortest
3Ø7Ø  LET B(Z) = I                          queues
3Ø8Ø  LET Z = Z+1
3Ø9Ø  NEXT I
31ØØ  LET R = 1+INT ((Z-1)*RND)
311Ø  LET I = B(R)                          select one
312Ø  LET N(I) = N(I)+1                     at random
313Ø  RETURN
```

As always, there's some initialization to take care of, and a bit of tinkering with the original program. First, DIM B...

```
19Ø  DIM B(9)
```

Next, we want to call either our old strategy at 16ØØ or our new one at 3ØØØ. That's easy enough:

```
38Ø  PRINT "CHOOSE STRATEGY:    1.RANDOM,"
39Ø  INPUT "    2.SHORTEST LINE. ( 1 OR 2 )";SNUM
```

Then we change the branch to:

```
545  IF SNUM = 1 GOSUB 1600 ELSE GOSUB 3000
```

Finally, we need to tidy the ANALYSIS:

```
2050  PRINT "CUSTOMER STRATEGY: ";("RANDOM" AND SNUM = 1)
      ("SHORTEST QUEUE" AND SNUM = 2)
```

With more than two strategies, this might get clumsy—*you* can work out a better way!

We now have an additional option in the simulation: the choice of strategy. Let's see what effect it has on the results. RUN as before; but when asked for the strategy input 2. Here's a sample with the same inputs as before: see how the results change!

STRAT = 2, AR = 3.

CN	ML	AL	MW	AW	AE
1	QUEUE FULL AT STAGE 14—RUN STOPPED				
2	10	5.55	35	14.657895	0.05
3	4	2.25	11	10.886364	0.05
4	5	2.6	16	9.9795918	0.65
5	3	1.57	10	7.3695652	0.5
6	3	1.4583333	11	5	1.15
7	2	0.69285714	7	5.7272727	2.95
8	1	0.4375	4	3.6666667	4.5
9	2	0.61666667	6	2.1698113	4

This time we get acceptable figures with only 4 or 5 queues. Note the dramatic jump in empty checkouts (AE) between CN = 5 and CN = 6. Again, the shortness of the run (20 stages, as before) may be causing distortions, so you should now try CN in the range 4-5-6-7 with, say, 100 stages. Experiment, and imagine you're the Manager deciding how many staff to hire. Or possibly the Consumer Watchdog, deciding whether the times are acceptable to customers.

DOCUMENTATION

The program is pretty much self-explanatory; but here's a summary of its main points.

When you run the program you will be prompted to supply the number

of checkouts, average rate of arrival of customers, desired length of run, and chosen customer strategy.

The program will perform the simulation, and provide an analysis of the results, listing the above variables, and also:

The maximum queue length.
The average queue length.
The maximum waiting time per customer.
The average waiting time per customer.
The average number of checkouts empty.

System variables

Numeric	AE	Average number of checkouts empty
	AL	Average length of queues
	AR	Customer arrival rate (number per minute)
	AW	Average waiting time
	CN	Number of checkouts
	EC	Counter in UPDATE routine
	ML	Maximum length of queue
	MQL	Minimum length of queue
	MW	Maximum waiting time
	NA	Number of customers arriving
	NSTAGE	Number of stages in simulation run
	NT	Next waiting time
	STAGE	Current stage in run
	SNUM	Customer strategy number
	TE	Running total of empty checkouts
	TL	Running total of queue lengths
	TCUS	Running total of customers arrived
	TW	Running total of waiting times
	V	ENQUEUE or DEQUEUE variable
Strings	D$	Dice—distribution of waiting times
	Q$	Used in QUEUE PRINT routine
Arrays	A (2)	Strategy line numbers
	B (9)	List of queues of minimum length
	H (9)	Head pointers for queue routines
	L (9)	Lap counters for queue routines
	N (9)	Length of queues
	Q (9, 25)	Queues
	T (9)	Tail pointers for queue routines
	W (9)	Total waiting times

Subroutine line numbers

1Ø	Initializations
3ØØ	Request for variables CN, AR, NSTAGE, SNUMB

5ØØ	One time step—main program
1ØØØ	Enqueue
11ØØ	Dequeue
12ØØ	Checkout graphics
13ØØ	Queue print
15ØØ	Arrivals
16ØØ	"Random" strategy
17ØØ	Next waiting time
18ØØ	Update
2ØØØ	Analysis
3ØØØ	"Shortest queue" strategy
9ØØØ	Autosave

Projects

If you've followed the instructions this far, you ought to be able to tinker with the program without much trouble; or even insert new subroutines allowing extra options (but be careful—you don't want to spend twenty minutes setting things up for each run!). Here are some suggestions—and some additional simulations you can try.

1. *Change the waiting times.* The "3Ø-sided die" D$ controls the distribution of waiting times. Change it, and see what difference it makes. Better still, let the computer build up D$ given the required distribution.

2. *Remove transients.* I've ignored almost completely the problem that we start with an "empty supermarket"—no customers—so the first few stages are not entirely representative. It might be more sensible to run the program for, say, 1Ø stages; reset all variables (except queues and related ones); *then* start the run proper. Try to write a program that allows this.

3. *Multiple runs.* You don't want to sit there all day keying in values 1, 2, 3, . . . for CN. Get the computer to do it; you'll need to store the various analyses in *arrays* now, and modify the final display. Oh, yes—and change the QUEUE FULL jump so that the machine does something useful, rather than just stopping.

4. *Coincident birthdays.* N people at a party compare dates of birth. Ignoring the year, what are the chances that at least two of them have the same birthday?

Simulate it. You'll need a 365-sided die (don't use a D$; use 365 * RND(Ø)). Don't worry about leap-years unless you're a perfectionist. Generate N birthdays, compare, count coincidences; repeat 5Ø times (say); print the results.

Try N = 5, 1Ø, 15, 2Ø, 25, 3Ø,. . . What size must N be fore the chances to be more than even (probability 1/2) that a coincidence occurs? Unless you know the answer, you'll be surprised. . .

5. *Gas station.* This has a line of PN pumps, each capable of accommodating two cars at a time, but serving only one. Arriving cars join one of two lines (for each side of the pumps), which they cannot leave until served. The lines move whenever a space on that side becomes empty. Given randomly distributed arrival and serve times, find maximum and average line lengths, waiting times, number of pumps idle, etc.

Generalize to PL lines, each with PN pumps.

6. *Hospital beds.* A hospital ward has BN beds. Patients arrive at random, and stay a random number of days. Beds are allocated as they become spare. If there are none, the patient goes on the waiting list. What is the optimum number of beds to ensure that the waiting list doesn't get too long, and that as few beds as possible remain empty on average?

If you think this is a thinly disguised rewrite of CHECKOUT—you're right! But the sensible numbers are different, and you'll need new graphics.

A case history in educational computing: test your French vocabulary and fly to the Moon at the same time!

10 French Countdown

by Eric Deeson, adapted by Cort Shurtleff

Eric runs EZUG, an educational software user's group. Here he writes of his experiences at the sharp end of program design and distribution, where theory must sometimes be temepered with practical considerations. So here, straight from the horse's mouth (no offense, Cort!) are some valuable tips on program design and a blow-by-blow account of their use; some special tricks to get more from your TRS-80 and some advice about selling and distributing software on the open market.

INTRODUCTION

Everyone knows that doctors at cocktail parties have a big problem. As soon as they mention their job, they are faced with a list of symptoms to diagnose. Teachers tend not to be invited to cocktail parties; however, they have been known to drop in on the occasional social gathering. As a computing teacher I have the doctors' problem – if another teacher finds out what I do, the reaction is likely to be, "Oh, could your write a program to help my 3C with their quantum mechanics?" Or the details of the Spartan wars. Or the two-times table, or whatever. My weak smile is sometimes followed by total inaction; sometimes an idea grows and reaches the stage of several scraps of paper. Very occasionally, after many hours' work, a polished program results. FRENCH COUNTDOWN came about in that kind of way. The story of how it reached a polished version can now be told. I would like to use it to illustrate how one can produce complex software by way of a methodical approach. This *is* a learning program, one that can be (and is!) used by children at home or at school. But the principles of structured program development are

generally applicable. There are also various coding tricks described, at least some of which you may find novel and useful.

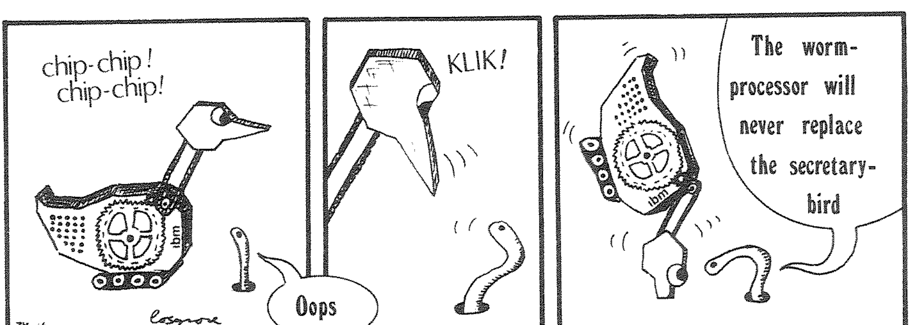
Any program, I think, has to meet the following criteria.

- It must do the job for which it was designed.
- It must do so efficiently and with minimum effort on the part of the user.
- Once the program is loaded, it should expect no computing knowledge at all. And the user should never be in doubt what to do.
- The screen display must be pleasing—effectively laid out, uncluttered, easy to follow.
- The user must enjoy using the program.

The designer needs, therefore, to know not only about programming, but about communication. And he/she must know the subject concerned. In the case of a teaching program, that theoretically means that three people form the development team—a competent programmer, a specialist in the teaching of the subject at the level concerned, and someone with an eye for language and layout. Brilliant as I am, I can't do all that for a French teaching program, and I consulted several teachers about different aspects of the software.

PROGRAM DESIGN

Teachers have to think on their feet a lot, fielding sudden questions with (one hopes) effective as well as immediate answers. I suspect, therefore, that computing teachers are more able than most programmers to write a workable program without much planning. However, that applies only to short chunks of code with straightforward objectives. The temptation to rush at a long complex program without thinking *must* be resisted. I've not been able to resist the temptation sometimes—and I've always got in a heck of a mess in the coding and not been satisfied with the



end result. We need a structured approach to the design and coding of programs likely to occupy more than a couple of dozen lines or so. A common type of structured development is called *top-down programming.* We can illustrate it like this:

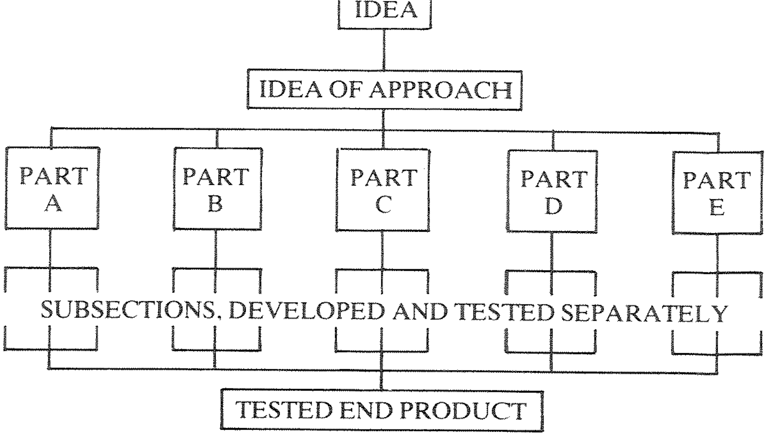


Figure 10.1

Note the word "tested" there. A major advantage of top-down programming is that one can test each individual procedure before linking it with the rest.

In essence, then, we must develop the initial idea into a number of fairly water-tight sections. Each section is called a *module*—it is developed, tested and polished on its own as a subroutine. The final program is a *suite* of such modules, suitably linked together and suitably tested and polished as a whole. Testing may seem irksome, for it is more than just trying the procedure yourself a couple of times to see if it works. The programmer must do his best to make sure it succeeds in *all* conceivable circumstances. If it doesn't what use is it?

The final program is therefore a set of modules, or subroutines, linked together in some suitable way. We can view it like this:

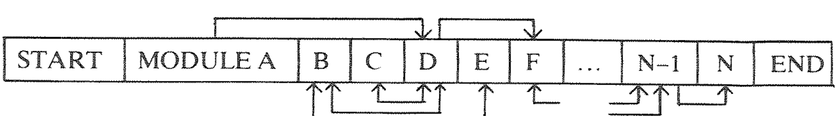


Figure 10.2

Actually I've been using words rather carelessly. It may not really matter here, but there are differences between subroutines, modules and procedures.

A *module* is a self-contained section of a program, one with a logical identity and which can be tested in isolation. A *subroutine* is a section of code designed to carry out a specific task. There are two kinds. The kind we are used to, called by GOSUB and terminated by RETURN, is a *closed* subroutine. An *open* subroutine, on the other hand, is a set of instructions in a larger one—it still has a specific task but does not use GOSUB/RETURN.

A *procedure* is a sophisticated closed subroutine with its own set of variables.

From now on, I'll use the word "module" to indicate a conceptual part of the final program. As far as coding is concerned, a given module may be a set of open and/or closed subroutines.

As I do quite a lot of coding, I find it useful to reserve different parts of the computer memory for different types of module. The TRS-80's memory map for my programs is laid out like this:

0	50	100	500	1000	1500	2000	5000	9998
REMS	START	INTRO	COMMON ROUTINES	MAIN PROGRAM	SPECIAL ROUTINES	DATA ROUTINES	GRAPHICS ROUTINES	SAVE

You'll see the differences between the sections when we get to look at the development of FRENCH COUNTDOWN.

Here are the stages of the structured modular development of a program. The stages marked * involve particularly careful testing, by the writer, by others, by victims representing the final target population.

1. Definition of program aims and objectives.
2. Development of plan of approach leading to outline flowchart and storyboard. (I'll explain these posh terms later.)
3. Analysis of the overall program into logical modules, based (theoretically) on the boxes in the outline flowchart.
4. Developing the main program module (*).
5. Developing, coding, and testing each module and subroutine.
6. Linking the modules together (*).
7. Revising as necessary (*).
8. Polishing the main and subsidiary modules for speed, layout, efficiency (*).
9. Adding the opening and closing routines (*).

10. Polishing and testing the whole thing.
11. Preparing documentation as necessary from the records developed earlier.

Records? Yes—keep the paperwork associated with steps 1-3 above, and during the coding stages maintain vigorously the following lists:

(a) Variables and their significance.
(b) Addresses of modules and subroutines.
(c) Graphics plotting data.

At the end, all this may be condensed into one page or so for filing—perfectly adequate if you keep your REMs in the listing.

Remember that no program is ever perfect—if it is worth keeping it is worth improving every so often. At the very least, you will come to find parts of it impossibly clumsy in the light of your growing expertise—those summary records in the file will make modification fairly straightforward.

DESIGNING FRENCH COUNTDOWN

By now you will have forgotten that I'm supposed to be talking about the development of a specific program—FRENCH COUNTDOWN, a language teaching unit.

How was all the theory of the last section applied in this case? Let's take it step by step.

Step 1: Definition of aims and objectives

The original cocktail party request was for a program with which individual pupils could test their elementary French vocabulary. That overall plan led to the following aims.

1. To draw at random from a pool of vocabulary items, requesting translation either from English to French or the other way.
2. To keep count of score.
3. To do all this within the context of a simple game.
4. To set up the pool so that it could be easy for the teacher/parent to change the vocabulary tested.

There was only one specific objective—that when the user had run the program two or three times he/she would know the vocabulary better. (That objective indicates that, in educational jargon, this is a *drill* program rather than one for just testing knowledge. I admit that the original request was for a simple test; trust me to think I know better.)

Step 2: Development of plan of approach

The above aims can be expressed in a very simple flowchart.

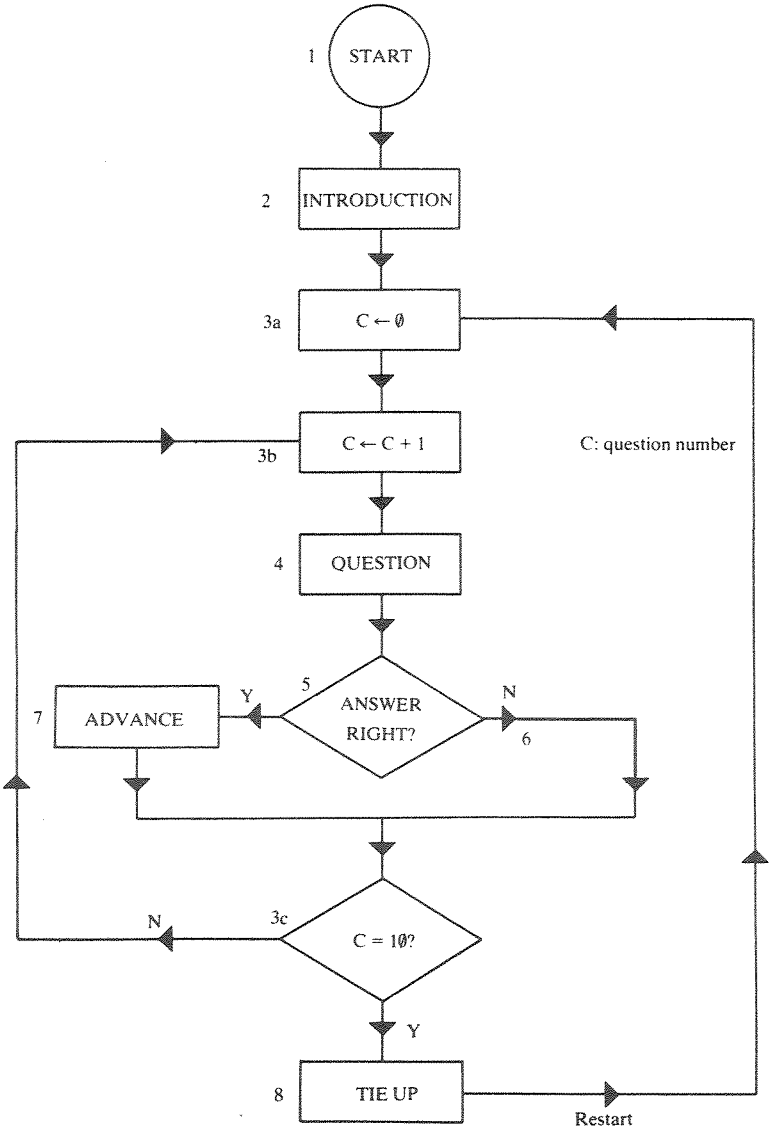
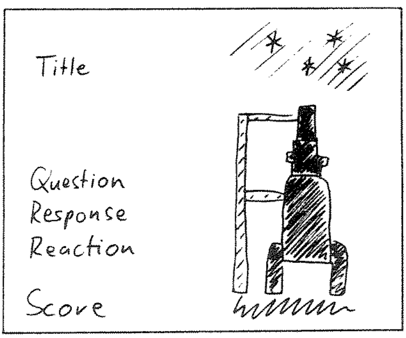


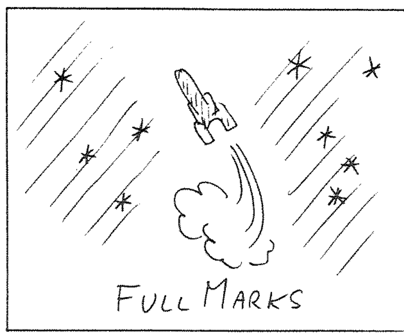
Figure 10.4

The skeleton of a set of modules is appearing out of the gloom already. Some are tiny routines, like the C-loop (setting a given number of questions); others, like getting an individual question onscreen, and checking out the answer, are going to be complex and will need much more definition.

What about that ADVANCE box? Therein lies the game aspect of this program. Some kind of "thermometer" display is needed, a graphics routine to show how successful the user is. Many programs of this nature, if they have a graphics game angle at all, stick to getting a train to move further and further along a track. Not me. With an eye on "Space Invaders" I settled on launching a rocket ship. Each correct answer would build the ship up further on the launch-pad, like sketch (a) below. Full marks at the end of the run, and the ship would buzz off around a screen full of stars. Sketch (b) shows that dream. (I didn't quite succeed in that, I tell you now.)



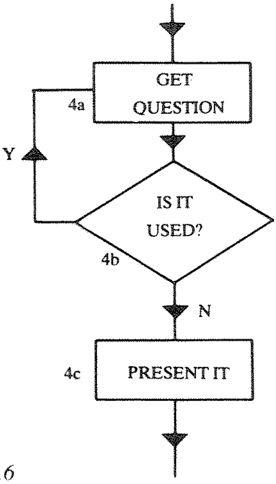
a



b

Sketches like this form what is called a *story-board* (a term intended to show my keen knowledge of the film industry, another industry in which I have failed to make a fortune).

Boxes 3, 4, and 5 are going to make the main program modules. 3 is simple—it needs no more extension as it will form a nice little FOR...NEXT loop. Box 4, though, needs more thought. Here the program must select a question from the pool, check that it hasn't already been used, and present it on screen. To some extent the routines there will depend on the data structure used to hold the pool of questions, but at this stage we can break box 4 down into smaller modules like this.



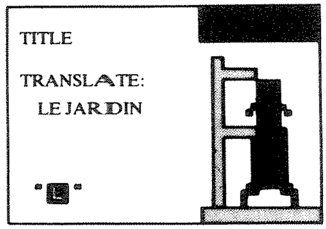


Figure 10.6

Maybe in practice 4a will need further subdivision, in particular to allow random choice of translation direction. Anyway, leave it for now. Same with 5, 6 and 7—they're likely to become complex in practice.

(Actually, at this stage of development, my plans for boxes 6 and 7 were—to say the least—vague.)

CODING

The foregoing took several hours, all without setting a single program line down on paper or in memory. Not only that, but those several hours were spread over several days. That is an accidental part of my program development procedures, for I do have other things to do than sit at a keyboard. But it is now an explicit part of my system. Even when I have a brilliantly exciting program idea I refuse to do any coding until a little while has elapsed in which my subconscious can kick routines around.

But now it's time to code! We're on to steps 4, 5 and 6 at last. First the barest skeleton, based on my standard memory map (sketched earlier):

```
   1  REM *** INITIALIZE ***
  35  REM *** START ***
 498  REM *** SERVICE ROUTINES ***
 999  REM *** PROG ***
1ØØ5  FOR C = 1 TO 1Ø
1Ø99  REM *** FINISH OFF ***
```

```
1169  REM *** BACK TO START ***
1498  REM *** SPECIAL ROUTINES ***
1999  REM *** VOCABULARY ***
4999  REM *** GRAPHICS ***
```

It's a start! And immediately some points arise:

1. These fragments of code are accompanied with (column 2) the growing list of variables.
2. Sprinkle REMs all over the place however good the paper-work. But give them line numbers just below the corresponding start-points—more on that later.

Having dealt with the skeleton I must now start building up the main program routine. You see I've allocated only a few lines to this (1ØØØ to 1Ø9Ø). (For FRENCH COUNTDOWN that proves to be insufficient—showing that I don't really follow my own rules too well.) Without a RENUMBER facility, which I tend not to like with any computer, that means cramped unidy line-numbers. Unless I run entirely out of line number space in a module, I try not to renumber by hand—it's time-consuming and unimportant even if it does make for a neater listing.

I've already dealt with box 3 in the flowchart, so let's get stuck into box 4. The first part of this, 4a, is "get question." As I said before, the procedure depends on how the question data are stored. By now I've decided to have sixty questions in the pool and I'm going to store them in two string arrays: FRNCH$ for the French words and then naturally, ENGLH$ for their English counterparts.

TRS-80 BASIC allows for an array depth (for us neophytes: that's the number of slots along one dimension of the array), without using a DIM statement, of eleven. Since I'm going to have sixty words, each of our two string arrays must have a depth of sixty. Clearly sixty is more than eleven, so I had to DIM each array (actually in testing the program I started out with an abreviated vocabulary of just ten words).

And while we're at it, such large string arrays all demand a CLEAR statement because left to its own devices the TRS-80 will reserve only 5Ø bytes of space to store strings in which is not enough for our purposes. We can get more space with the CLEAR n statement, where n is the number of bytes we want to reserve, but we must be careful for the statement will also set all variables to zero. Consequently, I put it at the beginning of the program where it will reserve the string space we need but won't erase anything since, at the start of the program, there isn't anything to erase yet.

What value for n? One character one byte, so 12Ø words of, say, length 12 are going to require 144Ø bytes of memory. To cover for other strings in the program and give a cushion I added an extra 5ØØ.

```
10  CLEAR 194Ø
15  DIM ENGLH$(6Ø) : DIM FRNCH$(6Ø)
```

The two arrays are, within a subroutine, filled with vocabulary after being called from the top of the program by a, you guessed it, GOSUB. The assignment coding,

```
1999  REM *** VOCABULARY ***
2ØØØ  FRNCH$(1) = "LA TABLE" : ENGLH$(1) = "THE TABLE"
2ØØ1  FRNCH$(2) = "LE TV" : ENGLISH$(2) = "THE TV"
2ØØ2  RETURN
```

was designed so that the vocabulary could be changed easily by editing out the old vocabulary and in the new. Note that I've only started with two entries—I'll expand the list later. Here's the subroutine call.

```
2Ø  GOSUB 2ØØØ
```

So let's enter the vocabulary fetch lines.

```
1Ø1Ø  CLS
1Ø15  PRINT@128,"TRANSLATE:";
1Ø2Ø  NUM = RND(2)
1Ø25  T = RND(2)-1
1Ø3Ø  IF T=1 THEN PRINT@139,FRNCH$(NUM);
1Ø35  IF T=0 THEN PRINT@139,ENGLH$(NUM);
1Ø4Ø  PRINT@256,"AFTER YOUR ANSWER,";@32Ø,"PRESS ENTER.";
```

Run it (with GOTO 1)—it works! By the way, I used those trailing semicolons on the PRINT@ statements to tell the machine that, apart from what I've had it print, I want the display left alone. If they weren't there it would erase anything previously printed on unused portions of the present line and, if that happened to be the bottom line of the screen, it would cause everything displayed to move up a line with embarrassingly amateurish results.

That leaves one tricky bit for the module of box 4—only allowing the subroutine to be accessed if it hasn't been used before. What we need is a "flag" system. If each routine starts off with its flag at Ø, and this switches to 1 when the routine is accessed, then we can test the flag and try again if it's 1. Here, an array is definitely the structure to use. I'll

set up an array U (for "used") with dimension 60; initialize each element to zero; test the flag; reject the choice of vocabulary if it's 1; accept the choice and set the flag to 1 if it's 0.

The initialization needs a subroutine of its own. Here it is:

```
  11  DIM U(60)
 100  GOSUB 1600
1599  REM *** ARRAY ***
1600  FOR N = 1 TO 60
1615  LET U(N) = 0
1610  NEXT N
1615  RETURN
```

Note how just before the beginning of the subroutine there is a REM statement that contains the subroutine's title. If, while looking over the program, I come upon a call for it and have forgotten what it is, all I have to do is list the line before the first line of the subroutine to jog my memory. I could have actually started the subroutine with the REM but by bypassing it I can speed the processing.

Right—when the program first runs, all those flags will become 0. Then testing for and conditionally setting the flags in the main routine:

```
1023  IF U(NUM) = 1 THEN 1020
1024  U(NUM) = 1
```

Get it? Fairly straightforward really. Now, put a temporary GOTO 1015 at line number 1045 and run it from the top. The routine should fetch a word once and then again You might need to add a delay routine beween fetches to see it. Check out the DELAY routine on page 94. Then it freezes up because both flag one and flag two will have been set and our random number generator, as presently structured, will only generate a one or a two.

Did you see that we have got a design bug to overcome? It might not have been evident if LE TV or THE TV was fetched before LA TABLE or THE TABLE, but as the program now stands old vocabulary words remain on the screen until they are written over. If a subsequent selection happens to be shorter than its previous counterpart, the remains of the first word will appear tacked onto the end of the second. This is not exactly what might be called polished. What I had to do was erase the first word before I fetched the next. So, to generalize the solution, at the start of the program I put in a variable for 12 spaces:

```
12  BLANK$ = "            "
```

and then whenever I needed it, I could use it, as in the following solution for our problem at hand:

```
1026  PRINT@139,BLANK$;
```

I must now admit that I decided upon the line number, 1026, only after some experimentation. You see there were a number of places on the screen where I eventually needed this "eraser" and it proved useful to erase almost the whole "board," as it were, with each go round of the PROG loop. 1026 fit in nicely as the spot to erase in. By the time coding was completed 1026 looked like this:

```
1026 PRINT@139,BLANK$;@460,BLANK$;@578,BLANK$;@333,BLANK$;
```

At this point I decided to expand the vocabulary table. So here we go again:

```
1999  REM *** VOCABULARY ***
2000  FRNCH$(1) = "LA TABLE" : ENGLH$(1) = "THE TABLE"
2001  FRNCH$(2) = "LE TV : ENGLH$(2) = "THE TV"
```

and so on up to the RETURN statement at line 2060.

But, hold it—if we've got twenty each of nouns, verbs and adjectives, can't we get a menu going? Of course we can. Mind you, it's going to mean messing about round 1020 again. Still—it'll be a nice feature.

```
405  GOSUB 800
410  PRINT@592,"I SHALL ASK YOU TEN QUESTIONS.";
420  GOSUB 500
425  PRINT@712,"WHAT TYPE DO YOU WANT THESE QUESTIONS TO
     BE?";
430  GOSUB 500
435  PRINT@860,"PRESS : ";@958,"   1 FOR NOUNS   2 FOR VERBS
     3 FOR ADJECTIVES   4 FOR MIXTURE";
440  INKY$ = INKEY$ : IF INKY$ < "1" OR INKY$ > "4" THEN 440
445  WRDTYP=VAL(INKY$)
450  GOTO 999
```

That calls two new service subroutines. I use DELAY in most programs. It causes execution to pause for a time set by the parameter DELAY. It can be cut short at any time, by pressing the spacebar in this case.

```
 45  DELAY = 75
499  REM *** DELAY ***
500  FOR B = 1 TO DELAY
```

```
5Ø5  INKY$ = INKEY$ : IF INKY$ = "" THEN GOTO 5Ø5
51Ø  IF ASC(INKY$) = 32 THEN RETURN
515  NEXT B
52Ø  RETURN
```

Yes, you're allowed two returns (or more) in a subroutine.

And here's TITLE, at 8ØØ:

```
8ØØ  REM *** TITLE ***
8Ø5  CLS
81Ø  B$ = CHR$(191)
815  FOR FOR I = 1 TO 18 : B$ = B$+CHR$(143) : NEXT I
82Ø  B$ = B$+CHR$(191)
825  PRINT@277,B$;@4Ø5,B$;@4Ø5,CHR$(143);@424,CHR$(143);
83Ø  PRINT@341,CHR$(191);" FRENCH COUNTDOWN ";CHR$(191);
835  RETURN
```

Before we get too far ahead of ourselves, don't forget to fill in the rest of the vocabulary table. Yes, all sixty entries and for our menu to work correctly enter 2Ø sets of nouns followed by 2Ø sets of verbs and finish up with the adjectives.

After all that convoluted rigmarole (sorry about it) we can LIST PROG and have a go at getting the value of WRDTYP in use.

```
1Ø16  RANGE = 2Ø : IF WRDTYP = 4 THEN RANGE = 6Ø
1Ø17  NUM = RND(RANGE)
1Ø18  IF WRDTYP = 2 THEN NUM = NUM+2Ø
1Ø19  IF WRDTYP = 3 THEN NUM = NUM+4Ø
1Ø2Ø  IF U(NUM) THEN 1Ø17
```

Follow that lot through with care. It may seem convoluted, but I think it's fairly neat in the circumstances. (OK, if I'd had the idea of choice of topics before I'd started, I'd have come up with an easier way—but I didn't.)

We now only have to deal with boxes 5-7 in the main program—checking if the answer's correct and acting accordingly. Actually we've been having rather a heavy time, so we'll do box 5 now with just a touch of 6 and 7 and then take a break.

Because of the random choice of translation direction (Lines 1Ø25-1Ø35) checking the correctness is not perfectly straightforward. The correct response is FRNCH$ if T is Ø and ENGLH$ if T is 1. So:

```
1045  INPUT R$ : IF R$ = "" OR LEN(R$)<3 THEN 1045
1050  PRINT@448,"YOUR ANSWER: ";R$;
1055  IF(NOT T AND R$ = FRNCH$(NUM)) OR
      (T AND R$ = ENGLH$(NUM)) THEN
      GOSUB 550 : ELSE GOSUB 600
```

For the moment let's not develop those two subroutines – we can just enter the following and test everything to date:

```
549  REM *** RIGHT ***
550  PRINT "RIGHT";
595  RETURN
599  REM *** WRONG ***
600  PRINT "WRONG";
645  RETURN
```

We'll need to spend quite a bit of time perfecting those routines – but later, after the break I promised.

GRAPHICS

This program is geting quite good already, but it's still only an automatic set-question-and-check-answer machine. The "countdown" in the title concerns the building and lauch of a rocket-ship. Remember? So now, to make a change, we'll go fairly fast through the graphics routines from line 5000.

We have ten questions in our main C-loop. So there must be ten stages leading to launch, each stage calling on the predecessors, and the stage reached depending on the score. Part of the RIGHT/WRONG modules will in fact determine score and call the countdown routines. We'll build the graphics up and test it bit by bit and as a whole.

Here's the overall plan.

```
4999  REM *** GRAPHICS ***
5049  REM *** GROUND ***
5095  RETURN
5099  REM *** SKY ***
5100  GOSUB 5050
5145  RETURN
5149  REM *** GANTRY ***
5150  GOSUB 5100
```

```
5195 RETURN
5199 REM *** BASE ***
5200 GOSUB 5150
5245 RETURN
5249 REM *** STAGE 1 ***
5250 GOSUB 5200
5295 RETURN
5299 REM *** STAGE 2 ***
5300 GOSUB 5250
5345 RETURN
5349 REM *** STAGE 3 ***
5350 GOSUB 5300
5395 RETURN
5399 REM *** CLEAR GANTRY ***
5400 GOSUB 5350
5445 RETURN
5449 REM *** COUNTDOWN ***
5450 GOSUB 5400
5495 RETURN
5499 REM *** LAUNCH ***
5500 GOSUB 5450
5595 RETURN
```

Can you see the pattern? Each routine calls on the preceding one, which calls on the one before, etc, etc. By the time the score reaches 10 you'll have given your subroutine stack a thorough work-out. This structure should be tested. Insert temporary lines like:

```
5050 PRINT "GROUND";
5105 PRINT "SKY";
```

and so on.

And then hold your breath and try direct entry: GOTO 5500. If the resulting screen activity is too fast to follow try, again temporarily, inserting our DELAY routine after each PRINT. Vary the size of the DELAY variable until the execution rate seems right for you. You should see an awful mess of messages building up—but they should all appear in the right order.

Now I'll reproduce the designs I came up with, stopping only for really necessary comments. Try to see my short-cuts; run each subroutine when coded by a direct GOTO whatever.

1. *Ground*

```
5050  GR$ = ""
5055  FOR I = 1 TO 25
5060  GR$ = GR$+CHR$(131)
5065  NEXT I
5070  PRINT@932,GR$;
```

2. *Sky*

```
5110  PRINT@43,"*";@100,"*";@124,"*";
5115  PRINT@169,"*";@184,"*";@315,"*";
```

3. *Gantry*

```
5160  PRINT@875,CHR$(170);CHR$(187);CHR$(149);
5165  PRINT@811,CHR$(170);CHR$(183);CHR$(181);
5170  PRINT@747,CHR$(170);CHR$(187);CHR$(151);
5175  PRINT@683,CHR$(170);CHR$(183);CHR$(149);
5180  PRINT@814,CHR$(176);CHR$(176);
5185  PRINT@750,CHR$(131);CHR$(131);
```

4. *Base*

```
5210  PRINT@879,CHR$(176);CHR$(176);
5215  PRINT@882,CHR$(176);CHR$(176);
```

5. *Stage 1*

```
5260  PRINT@880,CHR$(190);CHR$(143);CHR$(189);
5265  PRINT@816,CHR$(160);CHR$(176);CHR$(144);
```

6. *Stage 2*

```
5310  PRINT@816,CHR$(170);CHR$(191);CHR$(149);
5315  PRINT@752,CHR$(160);CHR$(176);CHR$(144);
```

7. *Stage 3*

```
5360  PRINT@752,CHR$(160);CHR$(191);CHR$(144);
5365  PRINT@689,CHR$(176);
```

8. *Clear gantry*

```
5410  PRINT@815,CHR$(144);@811,CHR$(186); @751,CHR$(129);
      @747,CHR$(171);
5415  GOSUB 500
```

```
5420  PRINT@815,“ ”;@81Ø,CHR$(16Ø);@751,“ ”;@746,CHR$(13Ø);
5425  GOSUB 5ØØ
5430  PRINT@814,CHR$(144);@81Ø,CHR$(176);
      @75Ø,CHR$(129);@746,CHR$(131);
5435  GOSUB 5ØØ
544Ø  PRINT@814,“ ”;@8Ø9,CHR$(16Ø);@75Ø,“ ”;@745,CHR$(13Ø);
```

9. *Countdown*

```
546Ø  FOR T = 9 TO 1 STEP-1
5465  PRINT@97Ø,“COUNTDOWN    T  MINUS ”;T;“ SECONDS  AND
      COUNTING”;
547Ø  GOSUB 5ØØ : NEXT T
5475  IF CFLG = Ø THEN PRINT@1007,“HOLDING  ”;
548Ø  CFLG = 1
```

The clear flag, set to zero at the beginning of each run of the program, is used so that the switch from printing “COUNTING” to “HOLDING” is not made if some knowledgeable or lucky student manages to reach LIFT OFF. After all, it wouldn't do to have “HOLDING” flash on the screen while the countdown continued down from 1 to Ø.

```
36  CFLG = Ø
```

Anyway it's off to launch we go. But, alas, not to realize my dream of having the rocket zoom off into star-set. BASIC's just too cumbersome. Here's how the launch ended up—quite majestic actually.

10. *Launch*

```
55Ø5  PRINT@992,“Ø”;
551Ø  TAIL = 944 : TIP = 753
5515  DELAY = 5Ø
552Ø  FOR I = 1 TO 16
5525  DELAY = DELAY-3*I : GOSUB 5ØØ
553Ø  TAIL-TAIL-64 : TIP = TIP-64
5535  IF I = 12 THEN 5555
554Ø  J = I-11 : IF J<Ø THEN 555Ø
5545  ON J GOTO 5555,556Ø,5565,557Ø,5575
555Ø  PRINT@TIP,CHR$(191);
5555  PRINT@(TIP+63),CHR$(17Ø);CHR$(191);CHR$(149);
556Ø  PRINT@(TIP+127),CHR$(186);CHR$(191);CHR$(181);
```

```
5565  PRINT@TAIL," * ";
5570  IF I<2 THEN 5580 ELSE PRINT@(TAIL +64)," * ";
5575  IF I<3 THEN 5580 ELSE PRINT@(TAIL +129),"  ";
5580  NEXT I
```

There it is, a rocket take off. Isn't it beautiful! Some might say it looks a lot prettier when you run it on the machine but us programmers know that the real beauty is in the logic of that crisp, well ordered and rational bit of coding, right?

If you didn't understand any of it, here are a few clues. Taking it from the top: the "Ø" finishes off the countdown; the TIP is the location of the tip of the rocket; likewise for the TAIL; with each pass through the loop the DELAY is shortened so that the rocket appears to accelerate. (For you physics buffs out there, here's a problem for you! Develop a set of equations which describe the acceleration, speed and postion of the rocket at time t. Hint: it reaches the speed of light very shortly after leaving the screen.) The loop draws, erases and redraws the rocket while shifting TIP and TAIL up the screen; the ON J GOTO jumps over successive parts of the rocket as it leaves the screen; and finally those last three lines before the NEXT I employ "*"s to create the spectacular special effects you see trailing the rocket.

A lot of care is needed when typing all that from a listing, so check it out properly—it's a nice rocket and it does take off.

POLISHING OFF THE MAIN ROUTINE

After all that work outside the PROG section, you may expect a lot of amendments within it. But no, that's the beauty of the modular approach. There's no effect on the center of the massive new building in the suburbs.

In fact, we've almost finished the main routine now. All that's left is to deal with the score (incrementing which is of course the function of the RIGHT subroutine).

We need to initialize the score:

```
1003  SC = Ø
```

and we need to display it in the C-loop:

```
1075  PRINT@704,"YOUR SCORE SO FAR";@768,"— — — — — — — — —";
1080  PRINT@833,SC;" OUT OF ";C;
1085  GOSUB 500
1090  NEXT C
```

And that's the end of PROG!

Just as well deal with 1Ø99 REM FINISH OFF now, too. That should make us feel good, though there will still be a few oddments left. Here's how I closed each run of FRENCH COUNTDOWN.

```
1100  GOSUB 800
1105  PRINT@646,"YOU HAVE HAD YOUR TEN QUESTIONS.";
1110  GOSUB 500
1115  PRINT "YOUR SCORE WAS";SC;".";
1120  GOSUB 500
1125  IF SC = 10 THEN PRINT@785,"WE HAVE LIFT OFF - WELL DONE.";
1130  IF SC < 10 THEN PRINT@787,"THE COUNTDOWN IS ON HOLD.";
1135  IF SC < 5 THEN PRINT@914,"YOU MUST LEARN YOUR WORDS.";
1140  DELAY = 150 : GOSUB 500
1150  GOSUB 800
1155  PRINT@655,"LET SOMEONE ELSE HAVE A TRY NOW.";
1160  GOSUB 500
1170  GOTO 35
```

Here's the OPEN routine the above calls on. You may guess from its position that in truth I set it up earlier in the development process than now!

```
1499  REM *** OPEN ***
1500  PRINT@785,"TO TAKE COMMAND PRESS ENTER."
1505  IF INKEY$ = "" THEN GOTO 1505
1510  RETURN
```

RIGHT OR WRONG

Now we come to the only remotely hairy bits left—developing the modules in subroutines 55Ø and 6ØØ. These deal, you recall, with correct and incorrect responses respectively, boxes 7 and 6 in the flowchart. For a correct response, we need:

(a) to add 1 to the score;
(b) a suitable message; and
(c) to build the rocket stage further.

I start off like this:

```
549  REM *** RIGHT ***
550  SC = SC+1
```

That's (a) done. Trouble with "suitable messages" is that they can be rather gruesome if they don't vary. So I decided to incorporate five suitable messages each for correct and incorrect responses, with random access to them. This called for two additional subroutines; RMSGS, for right messages, and WRMSGS for, you guessed it, wrong ones.

```
555  NUM = RND(4)
560  PRINT@578,RMSG$(NUM);N$;".";
```

RMSG$ is filled in during the start up as follows, subroutine call first:

```
  25  GOSUB 1700
1699  REM *** RMSGS ***
1700  RMSG$(1) = "RIGHT, " : RMSG$(2) = "GOOD, " : RMSG$(3) =
      "CORRECT, " : RMSG$(4) = "WELL DONE, "
1710  RETURN
```

Enter N$, the student's name, directly for testing purposes at this stage rather than restricting the later development of the introduction.

Now we can go on with our right answer coding...

```
565  FOR SPOT = 35 TO 995 STEP 64
570  PRINT@SPOT,BLANK$; BLANK$;"     "
575  NEXT SPOT
580  PRINT@970,BLANK$; BLANK$; BLANK$; BLANK$;
```

That erases the launch scene so that I could rebuild it with each right answer. Here's the call to the graphics sections:

```
585  ON SC GOSUB 5050,5100,5150,5200,5250,5300,5350,5400,5450,5500
590  RETURN
```

There's almost nothing at all to the wrong answer module except the same kind of message generator as used for right answers. Note the lack of an increment to the score.

```
600  REM *** WRONG ***
605  NUM = RND(4)
610  PRINT@ 578,WGMSG$(NUM);N$;".";
645  RETURN
```

And here are those wrong messages and the call to initialize them.

```
  30 GOSUB 18ØØ
1799 REM *** WRMSGS ***
18ØØ WGMSG$(1) = "NOT SO, " : WGMSG$(2) = "WRONG, " :
     WGMSG$(3)="NO,NO, " : WGMSG$(4)="SORRY, "
181Ø RETURN
```

See, I told you there was almost nothing to it.

THE END

The end is of course the beginning, the only bit we haven't entered. I suggest that that is the right idea—leave the introduction, especially the instructions, until the rest of the program is spotless. I don't think there's anything at all special about the remaining lines. So I'll just rush them off to you, and make one or two asides if need be. First a REM:

```
1 REM ERIC DEESON (C) 1982
```

And the introduction itself:

```
35 CLS
38 SC = Ø
4Ø GOSUB 15ØØ
45 GOSUB 8ØØ : DELAY = 75 : GOSUB 5ØØ
5Ø PRINT@528,"DO YOU KNOW YOUR FRENCH WORDS?";
55 GOSUB 5ØØ
6Ø PRINT@652,"IF YOU DO PREPARE FOR......LIFT OFF!";
65 DELAY = 15Ø : GOSUB 5ØØ : GOSUB 8ØØ
7Ø PRINT@531,"THIS IS MISSION CONTROL.";
75 DELAY = 75 : GOSUB 5ØØ
8Ø PRINT@652,"PLEASE TYPE YOUR NAME AND PRESS ENTER.";
85 INPUT N$
9Ø PRINT@783,"WELCOME TO THE LAUNCH PAD ";N$;".";
95 DELAY = 15Ø :GOSUB 5ØØ
```

And that, my friend(s), is that. Le programme (yes, that's what *they* call it!) est fini.

By now, of course, you've forgotten my opening pages. But rest assured that I've tried to meet the criteria my cocktail party friend needs:

- effectiveness;
- efficiency;
- ease of use;
- no computing knowledge required;
- no doubts as to actions required;
- carefully laid out, uncluttered, easy to read screen display;
- enjoyment – well, hopefully.

These were not in fact specifically educational criteria – I've also incorporated:

- full test details and corrections on hard copy;
- carefully determined (and varying) pace with delay-interrupt option;
- full mug-trapping;
- minimum predictability.

Well, it's helped *me* revise my French anyway. Et maintenant...

DISTRIBUTION

Having sweated many hours over a hot keyboard to produce an all-time masterpiece of useful programming, one's mind naturally turns to the possibility of making the material available to a wider audience than one's family or captive students.

There are many ways of distributing software in the hope of rich rewards. (They include sticking an ad in the computer press, submitting the material to a software library, or finding a publisher for it.) Whichever one chooses, there is now an extra stage – that of preparing the distribution version. This stage involves several activities – before any of which it is essential to get a full LISTing. These activities are:

1. Removing REMs – to save memory/loading time and to make it harder for others to follow.
2. Undertaking further techniques for cutting memory/loading time.
3. Polishing the whole thing to impress anyone who sees the coding.
4. Testing again, fully.
5. Renumbering (if you're keen), and testing *again*, fully.

There are, then, two major differences between the master version of a program and the one distributed. The first, and more important, is the minimization of memory requirement. Removal of the REMs so necessary during development, and shortening variable names can, in

particular, lead to a reduction of 10-15% in the length of the program and therefore in its loading and saving times.

The second aspect of preparation for distribution is, I suspect, the more important to those folk who're neurotic about copy-blocking. *There is no way to stop pirates copying your programs*. The method for copying any program is straightforward, but I shan't give it here. (Potential pirates are invited to send me—er—$50 for the secret, if they promise not to tell anyone else.) All the same most people distributing software *do* make some attempt to make it harder to pirate. They may at least make the listing hard to disentangle (well, that's the reason so often given for a cassette full of spaghetti), or they may prevent its being LISTed. If *you* are worried about this, best just "fingerprint" the program by including some dummy lines or directly entered combination code. Then at least you can test a suspected rip-off to see if it bears your fingerprints.

USING FRENCH COUNTDOWN

There are two schools of considered thought about user documentation. (I say "considered" because many people don't seem to give the matter any thought at all.) On the one hand there are the suppliers who feel duty bound to provide sheaves of literature with their programs; few users can follow it, let alone find it relevant. On the other hand, there are the folk who reckon that their progrms are self-documenting—and need no accompanying paper. I'm in the latter class. I have over 200 cassettes for the computer I primarily write software for (and as many again, in total, for the other computers I use). I find almost insurmountable the problem of storing accompanying paper so that it's as accessible as the tapes. And there are few things in life more frustrating than wanting to use a good program and not being able to find the instructions.

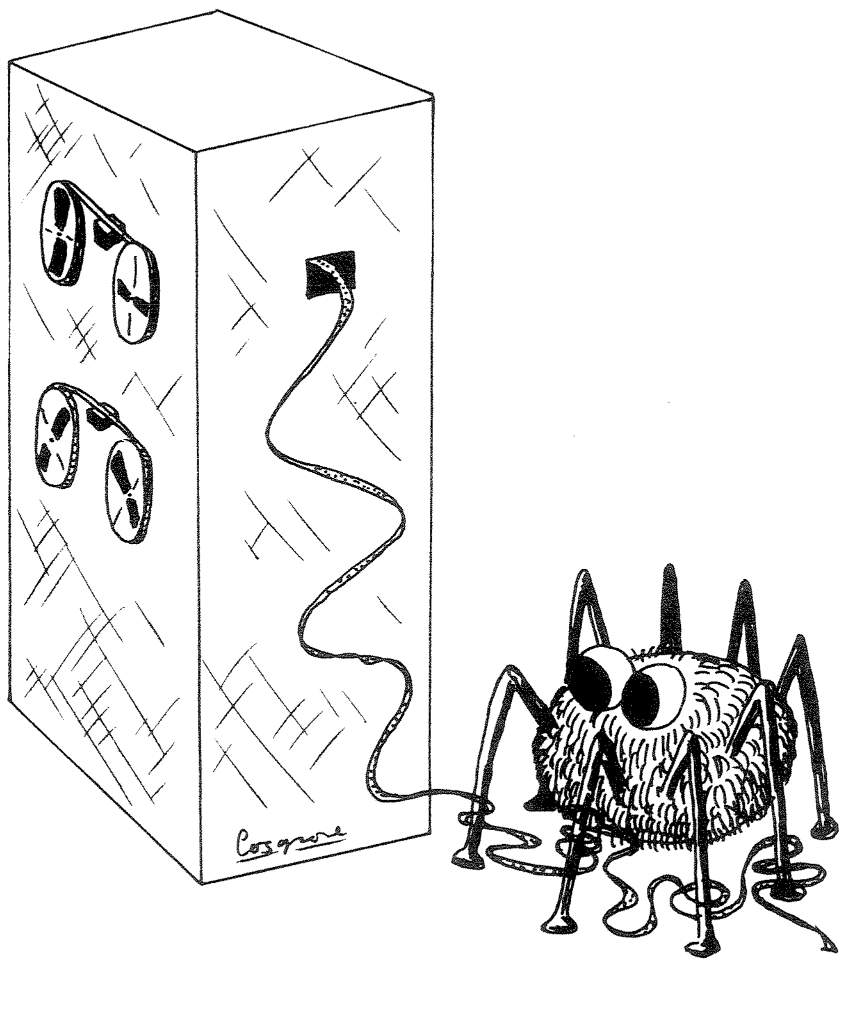
A fully self-documenting program must contain within itself all that the most inexperienced user is likely to need. That user is expected to know only how to get the software into the micro. So a program should be RUN when loaded and should never lead to a report code. Ideally the BREAK key should be disabled—in practice, I mask it stiffly enough so that it's very hard to actuate accidentally. Also the program should contain all necessary instructions. This takes up memory, of course—but that's a great inducement to keep the instructions simple. (Alternative approaches some people use are to have the instructions saved as a separate program on the tape, or to put an audio commentary on the reverse.) I think FRENCH COUNTDOWN is fully self-documenting. Well, almost. So here are all the user instructions it needs...

French Countdown recorded on cassette at higher than normal volume for the TRS-80 16K. Valid for any student after some six months of a school French course, this is a game for testing simple English/French and French/English vocabulary. The user is invited to select nouns, verbs, adjectives, or a mixture, and is presented with a test sequence of ten translations. Each correct answer prepares a rocket further for launch; the rocket takes off if all ten questions are answered correctly. Start with RUN. If delays are found to be excessive, speed them up using the space bar.

Teachers may wish to change the vocabulary used. It is stored from line 2ØØØ.

End of non-self-documentation.

Machine Code



An Odd Hexmas Tree

We've been able to tackle some pretty serious problems in BASIC, and I've never actually said anywhere, "This would be a lot easier if we could tackle it in language X," although I may have thought that a couple of times. So why worry with machine code at all? Won't it be much more difficult than BASIC? Is there anything to be gained?

The first question is of course rhetorical. The answer to the second question is that machine code isn't difficult to understand provided you have a clear grasp of the way the machine really handles data, and the form that the data take. The answer to the third is: "It depends." Let me elaborate:

There's a Z80 microprocessor (actually a Z80A, but this makes no difference) at the heart of your TRS-80 which does the real computing donkey work. Unfortunately, it only responds to cryptic, and very simple, instructions written in – you guessed it – its *machine code.* Any BASIC statement you want executed has first to be translated into this machine code, and that's done by a program (itself written in machine code) called an *interpreter,* which sits permanently in the Read Only Memory (or ROM) chip of your computer. This translation process takes time, and it's done every time the statement is executed. So if we bypass the interpreter, by writing directly in machine code, we get a dramatic improvement in speed. A program may run something like ten times faster! There are other reasons why speed improvements may be possible, but I'll leave those till later. Of course, whether this increased speed is worth the hassle depends on what you're trying to do. Some moving graphics displays may be hopelessly slow in BASIC. On the other hand, if you're just waiting for an answer to some complicated problem to be printed, you may be quite happy to sit around for 2Ø seconds rather than 2.

There can even be positive disadvantages to writing in machine code. A program *can* use more memory than its BASIC equivalent. (Again, we'll see why later.)

What I'm saying is that machine code is no cure-all. It's a tool, like any other, to be used in its proper place. If you've ever tried french – polishing a table with a chisel you'll know what I mean.

A QUICK CHECK

Since we'll need to wade through a fair amount of stuff before we can write our own machine code and understand what's going on, here's a couple of programs to demonstrate the power of machine language programming. They both do the same thing but with a difference. The first one is all BASIC, the heart of the second one is machine code stored within and executed by a BASIC program. Here's the first one. Type it in and RUN it.

```
 10  CLS
 20  DIM ADDS (3)
 30  ADDS(1) = 15630 : ADDS(2) = 15600 : ADDS(3) = 15830
 40  FOR I = 1 TO 3
 50  LET ADDRS = ADDS(I)
 60  LET LTR = 49
 70  FOR J = 1 TO 8
 80  FOR K = 1 TO (2*J - 1)
 90  POKE ADDRS,LTR
100  LET ADDRS = ADDRS + 1
110  NEXT K
120  LET ADDRS = ADDRS + 64 - 2*J
130  IF J = 5 THEN LET LTR = 64
140  LET LTR = LTR + 2
150  NEXT J
160  NEXT I
```

Now for the second one. Enter it exactly as listed. Machine language errors are notoriously unforgiving as you will undoubtedly discover in your pursuit of machine language expertise. If you're wondering about those DATA statements, they're just a convenient way of storing data, in this case our machine language routine, within a BASIC program. Do you see that READ statement further down in the program? Starting at the beginning of the first DATA statement, each time it's executed it pulls the next piece of data out and stuffs it into CODE. When the end of the first DATA statement is encountered it goes to the beginning of the next one and so on.

```
10  CLS
20  DATA 33,150,61,229,33,176,60,229,33,206,60,229
30  DATA 6,3,225,197,62,1,14,49,17,64,0,6,8,25
```

```
40  DATA 197,71,113,35,16,252,193,27,60,60,12
50  DATA 12,254,11,32,2,14,66,16,234,193,16,220,201
60  FOR I = 1 TO 50
70  READ CODE
80  LET ADDRS = 32255 + I
90  POKE ADDRS,CODE
100 NEXT I
110 POKE 16526,0 : POKE 16527,126
120 Y = USR(20)
```

Type RUN. Fast isn't it? It looped through the display in the same way the first one did. It's just that it's so fast that it appears to appear all at once. You'll never get that kind of speed out of BASIC.

When you've finished this book, you'll know that it's not magic at all: in fact you'll be able to write this kind of thing before breakfast. And you should be able to turn back to this page and answer two questions:

1. How does the machine language routine work?
2. What is the meaning of the obscure chapter title?

As I've told you before, the first thing is to understand the data structure. So what structure should *numbers* take in a machine code program?

11 Numbers in Machine Code

I said, a little while ago, that we were going to have to understand how the machine really represents data. Let's start with that.

We normally think about numbers in terms of tens. If I write the number 3814 we all understand that to mean:

3 × 1ØØØ + 8 × 1ØØ + 1 × 1Ø + 4 × 1

we can see that to get a "place value" from the one on its right we simply multiply by ten. We say the number is in *base* ten.

Because we've been doing this for as long as we can remember, it's difficult to realize that there are other, perfectly sensible, ways of doing the same job. Early computer designers certainly didn't; they used base ten representations in their machines and hit some nasty snags. Mostly they were caused by the fact that electronic amplifiers don't behave the same way for all the signals you want to input to them. For instance, an amplifier that is supposed to output double its input signal may well do so for inputs of 1, 2, 3, and 4 units; but then it starts to "flatten off" so that an imput of 5 produces an output of only 9.6, 6 produces 1Ø.8, and you can hardly tell the difference between the outputs for inputs of 8 and 9.

Put a music tape in your el cheapo cassette recorder and wind up the volume. Hear the distortion in the loud bits? It's the same effect.

Pioneer computer designers didn't hear any distortion; they just found that the machines couldn't distinguish between different digits at times, and that was hopeless for a computer. So they had to rethink their number representation to suit what the electronic dodads would do best.

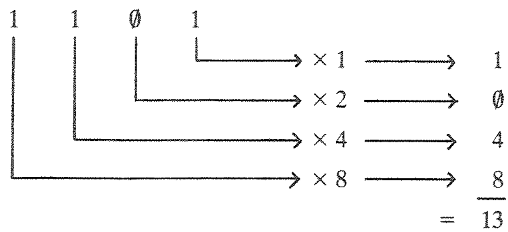
The simplest thing you can do with an electrical signal is to turn it

on or off; so you can represent the digits Ø (off) and 1 (on) satisfactorily. Distortion no longer matters. It's clear whether a signal is present or not regardless of how mangled it is. But can we devise a number system which only uses Øs and 1s?

Yes. In a base ten number, the largest possible digit is 9. Add 1 to 9 and you get 1Ø—a *carry* has taken place. We can write any number using any other base we choose, and the largest possible digit will always be one less than the base. If the base is 2, the largest digit is 1, so a base 2 (or *Binary*) number only contains Øs and 1s.

What about the place values? In the base ten case we got those by starting at 1 (on the right) and multiplying by 1Ø every time we moved left one place. For a binary number we still start at 1, but we multiply by 2 every time we move left.

So for instance the binary number 11Ø1 can be converted to base 1Ø like this:



Converting the other way is easy as well; take 25 for example. If we write down the binary place values:

32 16 8 4 2 1

and work from the left, it's clear that we need a 16, which leaves 9, and that's made up of an 8 and a 1, so 25 is:

Ø 1 1 Ø Ø 1

HEXADECIMAL CODE

This is fine for relatively small values, but a bit messy for large ones. There are a number of quick conversion techniques, but I want to examine a procedure which makes use of *hexadecimal* code, because it will stand us in good stead later.

A number in hex (nobody ever says "hexadecimal", except me, just now) is a number in base 16. So the place values are obtained by successive multiplications by 16. The first five are:

65536 4Ø96 256 16 1

"Hang on!" everybody's saying. "Those are nasty numbers, and anyway, in base 16 the largest digit has the value 15. Things are getting complicated."

Bear with me. We handle the problem of digits greater than 9 by assigning the letters A-F to the values 10-15. So the number 2AD in hex converts to decimal like this:

$$
\begin{array}{lllrl}
2\ A\ D & & & & \\
D & \longrightarrow \times 1 & \longrightarrow & 13 & (D \equiv 13) \\
A & \longrightarrow \times 16 & \longrightarrow & 160 & (A \equiv 10) \\
2 & \longrightarrow \times 256 & \longrightarrow & \underline{512} & \\
 & & = & 685 &
\end{array}
$$

Now for the nice feature of hex. Because 16 is one of the binary place values (the fifth one) it turns out that each hex digit in a number can be replaced by the four binary digits which represent it. (By the way, "binary digit" takes almost as long to say as "hexadecimal" so it's normally abbreviated to "*bit*".) The table below shows the conversions:

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

A more extensive table is given in Appendix 1.

Now suppose we want to convert 9Ø41 to hex. First we extract two 4Ø96s, then some 256s and so on like this:

```
                9Ø41
2 × 4Ø96 =  8192 -
            ----
             849
3 ×  256 =   768 -
            ----
              81
5 ×   16 =    8Ø -
            ----
               1
1 ×    1 =     1 -
            ----
               Ø
```

So the hex representation is 2351.

Now we just copy the digit codes from the table:

2	3	5	1
ØØ1Ø	ØØ11	Ø1Ø1	ØØØ1

and that's the binary equivalent of 9Ø41; just run the four blocks together to get 11Ø1ØØ11Ø1Ø1ØØØ1.

The hex-to-binary conversion is so easy that, more often than not, we leave numbers in hex even when, ultimately, we need them in binary. After all, it's easy to make an error in copying long strings of Øs and 1s.

CONVERSION BY COMPUTER

Here's a program to convert from decimal to hex. It successively divides the number by 16, looking at the remainder each time, so it extracts digits in the opposite order to that shown above.

Also it's going to use a few string functions which you'll find in the manual if you're not absolutely sure of them. Note also a minor annoyance built into ASCII code. Since character A does not immediately follow character 9, we've got to make the adjustment. And not for the last time either.

```
10  PRINT "DEC/HEX CONVERTOR"
20  PRINT "1) DEC - > HEX" : PRINT "2) HEX - > DEC"
30  PRINT "3) END"
35  PRINT "ENTER 1, 2 OR 3" : INPUT SEL
40  ON SEL GOSUB 80,170
```

```
5Ø   IF SEL = 3 THEN END
6Ø   PRINT " "
7Ø   GOTO 2Ø
```

The result is always presented as a 4-digit number. The program won't work if the result should contain more than 4 digits, but that's ideal for our purpose, as we shall see.

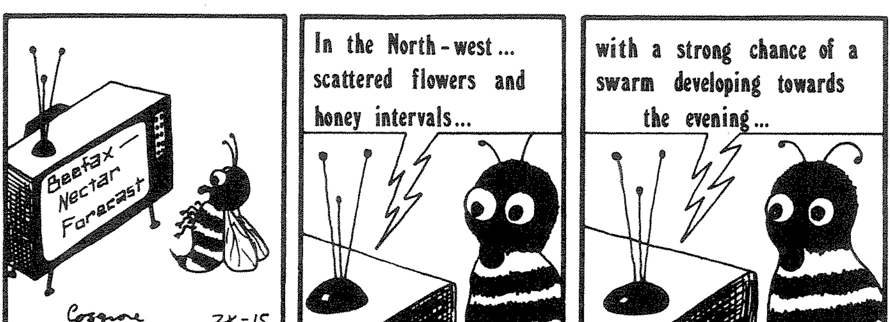
Here's the code to convert in the opposite direction (hex to decimal):

```
 8Ø   LET P = 4
 9Ø   PRINT "ENTER DECIMAL NO. (MAX = 65535)" : INPUT DN
1ØØ   LET N = INT(DN/16) : LET R = DN-16*N
11Ø   LET DISP = 48 : IF R>9 THEN LET DISP = 55
12Ø   LET HEX$(P) = CHR$(R+DISP)
13Ø   LET DN = N : LET P = P-1
14Ø   IF DN>Ø THEN GOTO 1ØØ
15Ø   PRINT "HEX VALUE IS ";HEX$(1);HEX$(2);HEX$(3);HEX$(4)
```

We could tie these routines together with a little menu:

```
17Ø   PRINT "ENTER HEX NO. (MAX FFFF, INCLUDE LEADING ZEROS)"
18Ø   INPUT HEX$
19Ø   D = 4 : DN = Ø : I = Ø
2ØØ   LET H$ = MID(HEX$,D,1) : LET N = ASC(H$)
21Ø   LET DISP = 48 : IF N>57 THEN LET DISP = 55
22Ø   LET N = N-DISP
23Ø   LET DN = DN+N*16 [ I
24Ø   LET D = D-1 : LET I = I+1
25Ø   IF D>Ø THEN GOTO 2ØØ
26Ø   PRINT "DECIMAL VALUE IS ";DN
```

and of course, we'll need RETURNs at lines 16Ø and 27Ø.



To deal with negative numbers, the machine uses a clever trick.

12 Positive and Negative

Now that we've seen something about manipulating binary numbers let's return to looking at the way they are handled inside the machine. Usually, a number is held in a fixed number of bits, often 16 or 24 or 32, depending on the machine design. This number of bits is called the *word size* for the machine.

Let's examine what numbers could be held in a 4-bit word:

4-bit pattern	Decimal value
ØØØØ	Ø
ØØØ1	1
ØØ1Ø	2
ØØ11	3
Ø1ØØ	4
Ø1Ø1	5
Ø11Ø	6
Ø111	7
1ØØØ	8
1ØØ1	9
1Ø1Ø	1Ø
1Ø11	11
11ØØ	12
11Ø1	13
111Ø	14
1111	15

It's obvious why bigger word sizes are chosen in practice; a machine which can only represent the numbers Ø to 15 is unlikely to be adequate. But there are two other problems; the notation can't represent fractional values (7.14, for instance) and it can't represent negative numbers.

We'll ignore the fractions problem because most machine code routines only use integers, but the way in which negative numbers are dealt with is more pressing.

The technique is simple: if you've got the binary representation of a positive number and you want to create its negative equivalent you do two things:

1. Change all the Øs to 1s and all the 1s to Øs (this is rather picturesquely called "flipping the bits").
2. Add 1 to the result.

For instance, suppose you want −3.

```
3 = ØØ11 in a 4-bit word

Flipping the bits gives:  11ØØ
Now add 1:                  +1
                          ----
                          11Ø1
```

So 11Ø1 represent −3. It's called the *2's complement* of ØØ11.

I'm not going to explain exactly why this works, but you can prove to yourself that it does in any particular case like this:

If we add 3 to −3 (or 5 to −5 or anything to minus itself) we should get zero. So:

```
     ØØ11      (= 3)
+    11Ø1      (= −3)
    -----
=   1ØØØØ
    -----
     111       (Don't forget that 1 + 1 = Ø carry 1 in binary!)
```

So we *don't* get ØØØØ at all; but the junior 4 bits *are* zero, and if we're working in a 4-bit word the senior bit will just drop off the end. (For a convenient analogy, think about a car trip-meter with 3 digits; if it reads 999 and you drive an extra mile, it reads ØØØ and a "1" has "dropped off" the left hand end).

In other words we should have seen it like this:

```
    [ØØ11]
+   [11ØØ]
    ------
 ┌──[ØØØØ]
 ↓
 1
```

This always works provided that the number of bits is fixed throughout. Don't forget to include leading zeros to make up the number of bits to this standard length, *before* taking the 2's complement.

Let's rewrite the 4-bit table of values, now including negatives:

Decimal	Binary	2's complement	Decimal
Ø	ØØØØ	ØØØØ	Ø
1	ØØØ1	1111	−1
2	ØØ1Ø	111Ø	−2
3	ØØ11	11Ø1	−3
4	Ø1ØØ	11ØØ	−4
5	Ø1Ø1	1Ø11	−5
6	Ø11Ø	1Ø1Ø	−6
7	Ø111	1ØØ1	−7
8	1ØØØ	1ØØØ	−8
9	1ØØ1	Ø111	−9
1Ø	1Ø1Ø	Ø11Ø	−1Ø
11	1Ø11	Ø1Ø1	−11
12	11ØØ	Ø1ØØ	−12
13	11Ø1	ØØ11	−13
14	111Ø	ØØ1Ø	−14
15	1111	ØØØ1	−15

Straight away we see that there's a problem; every bit-pattern occurs twice so that, for instance, 1ØØ1 could mean 9 or −7. So we'll have to restrict the range of values still further. I've drawn a dotted line around the region we actually choose to represent. If you look at the senior (leftmost) bit in each of the patterns you'll notice that it's "Ø" if the number is positive and "1" if the number is negative. This is obviously a very convenient distinction.

So the range of numbers we can get into a 4-bit word is −8 to +7. For 5 bits it would be −16 to +15. For 6 bits it will be −32 to +31 and so on.

A 16 bit word (which is important so far as the Z80 is concerned) holds the range −32768 to +32767. A table of 2's complement notations for 8-bit words is given in Appendix 1.

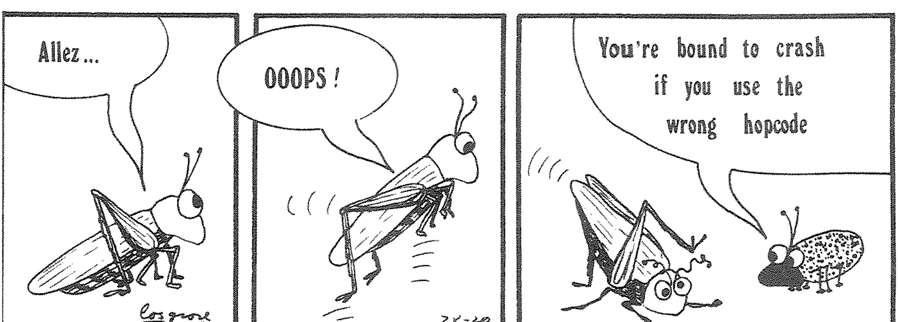
It's easier to start with a simplified, imaginary machine.
The Z80 is like this, but more complicated:
get the main ideas here!

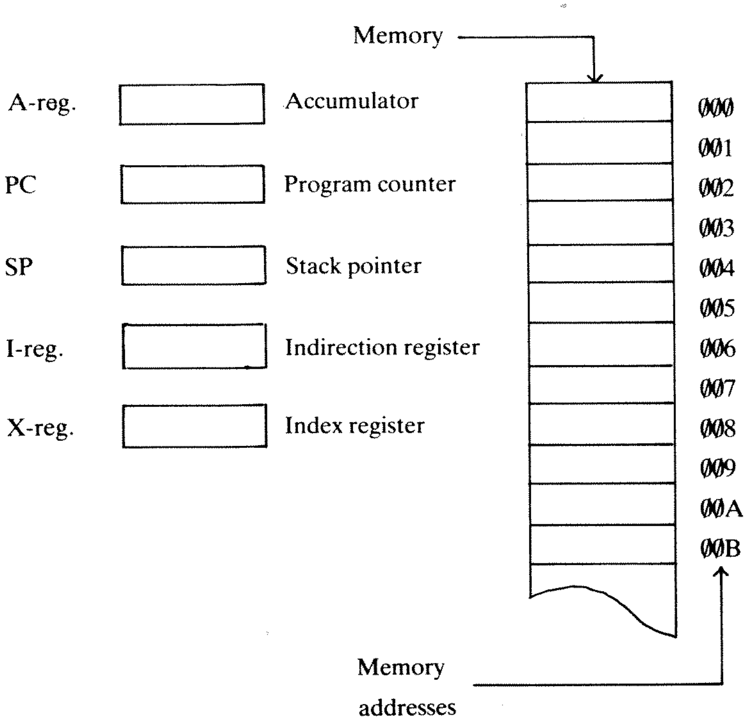
13 Machine Architecture

That's enough about numbers. Now we'll look at how the machine crunches them. To do this, we need to know about the internal structure of the processor – its *architecture*.

Now, the Z80 processor is the product of some twenty-five years of computer development and is a fairly sophisticated beast. So it's not really a good place for the beginner to start. What I'm going to do, then, is describe a simple processor which might have been built in the late 1940s (except it wasn't), just to introduce the important concepts which are relevant to virtually all current devices, without having to worry about the frills, which we can look at later (in Chapters 16 onwards).

We'll suppose that our imaginary machine has a memory of 16-bit words and a number of 16-bit special-purpose registers as shown below:





Let's look at the memory first. In BASIC we could have called each of those memory locations anything we fancied, but the naked machine isn't so friendly. It insists on numbering every location in an absolutely fixed way, starting at zero, as I've shown. These numbers are called the *memory addresses*, and I've numbered them in hex, although you should always bear in mind that, ultimately, the coding will be binary.

What can be held in a memory word? Well, any pattern of 16 bits. Obvious; but the point I'm driving at is that those 16 bits can mean anything we want them to mean. If we want them to mean a 2's complement coded integer then a word holds a number in the range −32768 to 32767. If we want them to mean a positive integer with no sign bit then the number is in the range Ø to 65535. If we want, we can split the word into two 8-bit fields each of which represents an alphabetic, punctuation or graphics symbol. As Tweedledee (or was it Tweedledum?) said: "When *I* use a word it means just what I choose it to mean— neither more nor less." I sometimes think Lewis Carroll was ahead of his time.

Now for the special-purpose registers. Just the A-register to kick off with. This is used every time you do any arithmetic. The result of any

sum you ask the machine to do is put into the A-register. (Sometimes it's called the *accumulator*, by the way.) Most arithmetic operations work on two values; it's no good asking the machine to work out 3+, you need to say what 3 is to be added to. One of these values must be in the A-register before the addition operation is executed. So you can write an instruction like:

```
ADD (1A3)
```

and the machine takes that to mean:

1. Add the contents of memory location 1A3 to the contents of the A-register. (The brackets round 1A3 are being used to indicate that it's the *contents* of 1A3 and not the *number* Ø1A3 which is to be added.)
2. Put the result back in the A-register.

We've just written our first machine level instruction. It's not actually in machine code, but it's close. Look at its general form. It consists of an operation code, ADD, and an address, (1A3). Many instructions will look like that. Incidentally, life is too short to say "operation code" too often; everybody shortens it to *opcode*.

AN ADDITION PROGRAM

Let's think about a sequence of machine instructions which would model the BASIC statement:

```
LET R = B + C
```

First we would have to assign actual addresses to R, B, and C. Suppose that these are 1Ø3, 1Ø4 and 1Ø5, respectively. We have to get the contents of 1Ø4 into the A-register. Let's invent an LD (for load accumulator) instruction to do this:

```
LD (1Ø4)
```

then add on the contents of 1Ø5

```
ADD (1Ø5)
```

and finally we need a way of storing the A-register's contents back in 1Ø3. So we'll invent a "store" instruction:

```
ST (1Ø3)
```

Now we have a simple machine level program consisting of 3 instructions:

```
LD (1Ø4)        [load B into A-register]
ADD (1Ø5)       [add on C]
ST (1Ø3)        [put the result in R]
```

How do we get the machine to run such a program?

We're used to the idea that a program is stored in the machine *before* it's executed. After all, if you wrote the BASIC statement:

```
1Ø PRINT "HELLO WORLD"
```

you'd be somewhat disconcerted if, as soon as you hit ENTER, the message "HELLO WORLD" were displayed. You expect it to be held until you need it. So, by the same token, a machine level program has to be stored first. Where more natural to store an instruction than in a memory word? (A word means what you want it to mean—remember?) Of course, that implies that the opcodes LD, ADD and so on have to be coded as bit patterns, but all we have to do is invent a table of bit patterns in a quite arbitrary way like this:

Opcode mnemonic	Binary code
ADD	ØØØØ
LD	ØØØ1
ST	ØØ1Ø

and every time we think of a new opcode that's needed, we add it to the table.

I've assumed, above, that all opcodes have a 4-bit binary code. That allows 16 different patterns and therefore 16 distinct instructions. This is a small instruction set by modern standards, but it will do for our hypothetical toy computer. We've got 16 bits in the word altogether, so 12 are left for the address portion of the instruction.

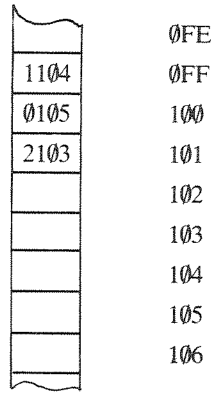
So LD (1Ø4), once inside the machine looks like:

$$\underbrace{\boxed{Ø\ Ø\ Ø\ 1}}_{\text{opcode}}\underbrace{\boxed{Ø\ Ø\ Ø\ 1\ Ø\ Ø\ Ø\ Ø\ Ø\ 1\ Ø\ Ø}}_{\text{address (1Ø4 hex converted to binary)}}$$

Once you've seen one bit pattern, you've seen them all, so from now on we'll write the hex versions of instructions. It's marginally less tedious.

THE PROGRAM COUNTER

Suppose we store our 3-instruction program from location ØFF onwards:



Now we need a way of saying to the machine: "Kick things off by executing the instruction of ØFF, then do the one in 1ØØ, then one in 1Ø1." That's what the PC-register, or *program counter*, is for. It acts as a kind of bookmark for the computer. We run the program by initializing the PC to the address of the first instruction. While the machine is obeying this instruction, the PC is automatically updated by 1, so that when the system returns to examine the PC, it will go and obey the next instruction, and so on.

There's a snag, though. While the last instruction (in 1Ø1) is being dealt with, the PC will be updated by 1 as usual, and so when the machine looks at it again, it will find 1Ø2, and leap off to execute the instruction there. What instruction? We didn't put one in 1Ø2. Ah! But there has to be a bit-pattern in 1Ø2 left by a previous program, or just set up when the machine was switched on. So the machine will interpret this pattern as if it is an instruction, because that's what we've asked it to do. And then it will roll on through locations 1Ø3, 1Ø4, and 1Ø5 and that's where we're storing data! So if the number in 1Ø4 is 2ØFF, for instance, the machine will interpret this as:

```
ST (ØFF)
```

which will copy the contents of the A-register into ØFF, thereby destroying the first instruction of our program! Obviously what we need is a "halt" instruction (I'll use the mnemonic HLT) which stops the updating of the PC in its tracks. So the program now reads:

```
LD (1Ø4)
ADD (1Ø5)
ST (1Ø3)
HLT
```

There's an important point here. Precisely because we are using words to mean different things at different times, we have to keep a very careful eye on the implications the machine will draw from what we tell it to do. If we request it to ADD the contents of a location to the A-register, then it will assume that that location holds a number. It will make no tests; it cannot—any bit-pattern could represent a number. Similarly, any bit-pattern could represent an instruction, so if the PC points to a location, its contents will be executed as an instruction.

The rule is: *keep data and programs firmly apart.* If you don't you can expect to be totally mystified at regular intervals. As I've indicated, a whole program can disappear without trace while it is running!

Some more instructions: the functions of the program counter and the stack.

14 Jumps and Subroutines

So far, our instruction set looks a bit thin. We've got LD and ST, which will move things around memory, ADD, which is pretty primitive arithmetic, and we can stop things with HLT.

We'll pep up the arithmetic capability a bit by adding SUB, which will subtract the contents of a location from the A-register, but that's all we're getting. No multiply, no divide, definitely no square root.

What we really need is a set of branch instructions, equivalent to BASIC's IF...THEN...

JUMPS

It's going to be fairly easy to branch to an instruction out of the usual sequence; what we need to do is change the contents of the PC. So we'll use an instruction like:

```
JP 416        [jump to 416]
```

Whenever it is executed, it will put 416 in the PC. The system is "fooled" into thinking that the next instruction is in 416, and then it will go on to 417, 418 etc. until the next "jump" instruction is encountered. Of course, any address can follow the JP code.

This instruction is more like a GOTO than an IF...THEN ... What we need is an instruction which resets the PC only if some condition is met. The simplest test we can make is whether the A-register contains zero.

```
JPZ 2A7       [jump to 2A7 only if A-reg. contains Ø]
```

Another would be:

```
JPN 14E       [jump to 14E only if contents of A-reg. are negative]
```

That's the minimum we can get away with, because we can now test for a positive (non-zero) number by noticing when the program doesn't jump on either JPZ or JPN instructions.

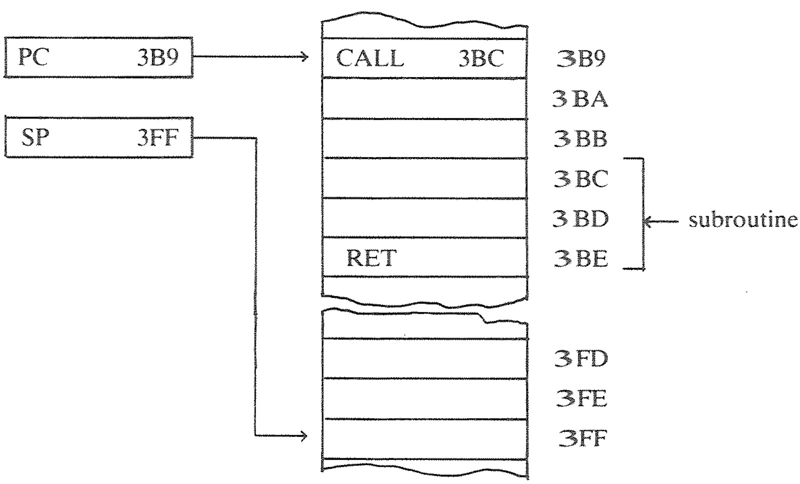
SUBROUTINES AND STACKS

While we're on the subject of transferring control from one place to another inside the program, how about something like BASIC's GOSUB and RETURN?

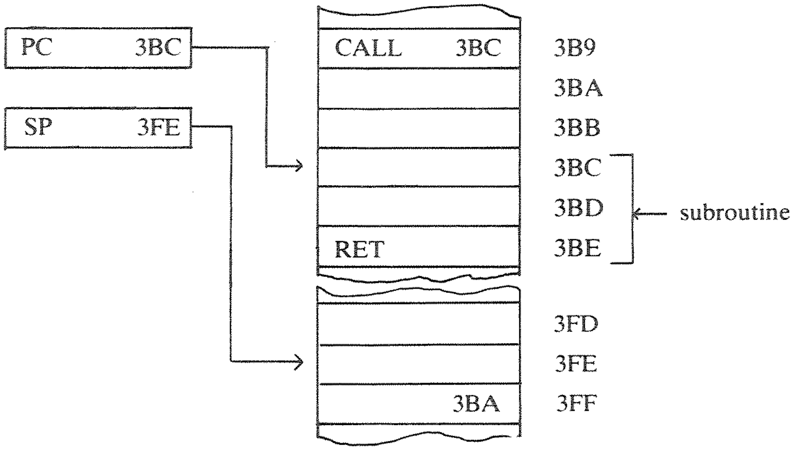
We'll have an instruction:

CALL 2Ø5 [call the subroutine starting in 2ØØ]

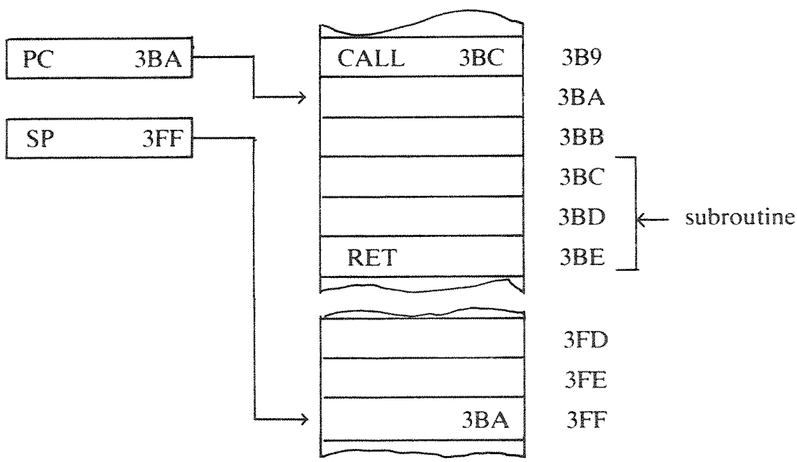
What does it do? Well, obviously it puts 2Ø5 into the PC, but we could use a JP for that. CALL performs a second function: it stores the address of the instruction after the CALL, so that when a "return" (opcode: RET) is encountered it can load the stored address back into the PC to continue the main program from where it left off. This is where the SP register comes in. We use some of the memory as a *stack* (remember stacks?) and SP points to the top of the stack. When a CALL is obeyed, the return address (the address of the CALL + 1) is pushed on to the stack. When the RET is encountered the stack is popped into the PC. Here's an example:



The CALL is about to be obeyed...



Now it has been, and the return address is on the stack. The program steps through the subroutine until it reaches the RET, after which:



and control is back inside the main program.

The private eye tracks his victim: how
to use the contents of one address
to point to another one.

15 Indirection and Indexing

There are only two registers left to talk about, and both have similar functions: they can both alter the address part of an instruction while the program is running.

INDIRECTION

Let's look at the way the I-register does this first. We'll invent a new opcode, LDI or "load indirect." Like HLT, it doesn't have an address associated with it. To the machine, it's just like an LD except that the high bit of the address field is set to "1." This bit is called the *indirection flag,* and simply indicates to the machine that indirection is in force. So the binary form of the LDI instruction is:

Ø Ø Ø 1	1	Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø
opcode	indirection flag	address (not used)

The hex code is 18ØØ. When the machine encounters this instruction, it uses whatever number is in the I-register as the effective address. So if the I-register contains 1E4 and an LDI instruction is executed, the effect is exactly the same as if the instruction had been LD 1E4. In other words, the I-register acts as a memory pointer, and we can move it around to our heart's content if we can do arithmentic with it. That means moving values into the A-register, because that's the only place we can do arithmetic. So we'll invent an opcode XAI for "exchange contents of

A-register with contents of I-register."

Of course, the indirection flag can be set for any instruction which has an address part. So we can have STI, JPI, ADDI etc. and in each case, the last 3 digits of the hex code will be 8ØØ.

AN EXAMPLE

Let's look at an example which uses these ideas. Suppose that we want to initialize a 1D array of length 2Ø, to hold the numbers 2, 4, 6, 8. . . 4Ø. In other words we want a machine code equivalent of the BASIC:

```
FOR C = 1 TO 2Ø
LET A (C) = C*2
NEXT C
```

There are a series of values which are going to have to be in memory somewhere, to make this work. They are 1 (because the loop count goes up in ones), 2 (because that's the increment for the array contents) and 2Ø (which is needed to test for the end of the loop). I don't, for the moment, want to be bothered with exactly where these numbers should be stored, so I'm going to allow addresses to be referred to temporarily by names (just like BASIC names). We'll have to convert these to numbers when we finally get to machine code, of course. This is an application of Jones's First Law of Computing: "Never put off till tomorrow what you can put off till the day after." So we'll assume that the numbers we want are available in locations called N1, N2 and N2Ø. Similarly, we'll have a location called BASE which holds the address of the first element of the array, and one called COUNT which will act as the loop counter.

First we set the I-register to point to the base of the array:

```
LD  BASE
XAI
```

Then we set the COUNT to 1:

```
LD  N1
ST  COUNT
```

Now we double this (by adding it back into the A-register) and store it in the location pointed at by the I-register. (We talk about "storing *through* the I-register" for short.)

```
ADD  COUNT
STI
```

We "undouble" the value on the A-register again, subtract 2Ø and see if the result is zero. If it is we've finished:

```
SUB  COUNT
SUB  N2Ø
JPZ  OUT
```

OUT is another, as yet unspecified, address. We don't know where it is yet, because we don't know where the program ends, and so, again, it's useful to give it a name temporarily.

If the branch doesn't occur, we add 1 to the COUNT:

```
LD   COUNT
ADD  N1
ST   COUNT
```

and increment the I-register by 1:

```
XAI
ADD  N1
XAQI
```

The current COUNT is now back in the A-register, so we can loop back to the doubling operation:

```
JP  LOOP
```

provided we give the "ADD COUNT" instruction the symbolic address "LOOP." Let's do this by preceding the instruction by its symbolic address followed by a colon:

```
LOOP:  ADD  COUNT
```

We can do the same sort of thing to set up the initial values we need, by defining a new opcode HEX which just sets a word to a required value. It isn't really an opcode at all since it isn't equivalent to a machine instruction, so we call it a pseudo-operation. The whole program looks like this (ignore the numbers in the left- and right-hand margins for the moment):

```
Ø2Ø           LD     BASE    1 Ø33
Ø21           XAI            A ØØØ
Ø22           LD     N1      1 Ø3Ø
Ø23           ST     COUNT   2 Ø32
Ø24 LOOP:     ADD    COUNT   Ø Ø32
Ø25           STI            2 8ØØ
Ø26           SUB    COUNT   4 Ø32
Ø27           SUB    N2Ø     4 Ø31
Ø28           JPZ    OUT     6 Ø47
Ø29           LD     COUNT   1 Ø32
Ø2A           ADD    N1      Ø Ø3Ø
Ø2B           ST     COUNT   2 Ø32
Ø2C           XAI            A ØØØ
Ø2D           ADD    N1      Ø Ø3Ø
Ø2E           XAI            A ØØØ
Ø2F           JP     LOOP    5 Ø24
Ø3Ø N1:       HEX    ØØØ1    Ø ØØ1
Ø31 N2Ø:      HEX    ØØ14    Ø Ø14
Ø32 COUNT:    HEX    ØØØØ    Ø ØØØ
Ø33 BASE      HEX    ØØØØ    Ø ØØØ
```

The only symbolic address which doesn't appear in the left-hand column, and is therefore still unspecified, is OUT. We'll worry about it later.

The form of the program we now have is written in what is known as *assembly code*. On modern sophisticated computers there will be an *assembler program* whose function is to convert this into real machine code for us.

Assemblers have been written for the TRS-80 and are available on both cassettes and disks. However, we're going to learn a lot more about computers and about assembly language basics by hand assembling our routines. If, after you've gotten a feel for machine language programming, you think it meets your programming needs, you should consider buying an assembler. They're surprisingly cheap and with what you've learned here you'll have no problem getting right in over your head. Of course if you're a purist (masochist) you should do all your programming in machine language (binary that is, hex is too human).

HAND ASSEMBLY

First we need a table of opcodes and their equivalent hex values:

Opcode	Hex
ADD	Ø
LD	1
ST	2
HLT	3
SUB	4
JP	5
JPZ	6
JPN	7
CALL	8
RET	9
XAI	A

Also we need to know where the beginning of the program is. That's a more or less arbitrary decision, so let's assume it's at Ø2Ø. Since each instruction occupies 1 word, we can write down the address of each instruction. You'll see that I've done this down the left-hand side of the program. Now we can replace the opcodes and addresses by their hex equivalents. For instance, LD BASE becomes 1Ø33, since BASE is now identified as Ø33. The right-hand margin shows the complete code.

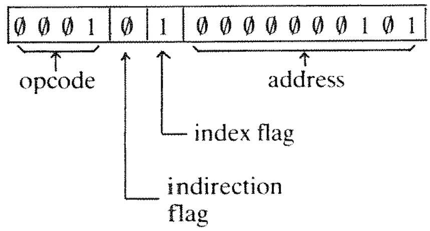
The only instruction which needs further comment is JPZ OUT, which encodes as 6Ø47. Why should OUT be at Ø47? It could be elsewhere, but Ø47 is the first location it can be at. The reason is that the array is occupying the space from Ø33 to Ø46 (twenty words), and we obviously don't want to go clumping around inside the program's data area.

THE INDEX REGISTER

When the X-register is in use, the real instruction address is formed by adding the address field to the contents of the X-register. For instance, if the X-register contains 4ØØ, then the instruction LDX ØØ5 has the same effect as LD 4Ø5.

We'll pinch another bit of the address field to indicate when indexing

is in operation, so the LDX ØØ5 instruction looks like this:



In hex, that's 14Ø5.

Actually there's nothing you can do with indexing that you can't do with indirection. It's just that it will do arithmetic with addresses automatically instead of leaving the job to you.

The actual architecture of the Z80 CPU,
the heart (or is it the brain?)
of your TRS-80.

16 At last the Z80!

I'm sorry that you've had to wade through the last ten or so pages without being able to try anything out, but if you've really understood the ideas in them, you'll find that understanding the Z80 is a breeze.

Before we get into the Z80's architecture (sorry, the chapter heading isn't quite accurate!) let's consider some of the difficulties of the processor I've just described.

First, the 4-bit operation code only allows 16 different instructions. (OK, we cheated a little, by allowing the indirection and indexing flags to spill over into the address field, but that in turn means we've limited the address size, and therefore the maximum size of memory!) The Z80 has 694 instructions! To give each of them a separate bit pattern means that we need an 8-bit field (1 byte); and even then some fudging is needed.

Second, our imaginary machine uses memory in a rather careless way. Some of the instructions don't use the address field (HLT, LDI, STI, for instance), so a sequence of such instructions wastes 1Ø bits in every word. The Z80 gets over this problem by allowing different instructions to have different lengths. Some instructions have no address field and are just 1 byte long. Others have a 1-byte address field and so are 2 bytes long. Others have a 2-byte address field for a total of 3 bytes. There are even some which have 2-byte opcodes! This means that the PC can't increment by 1 for every instruction executed. It has to increment by the length of the instruction.

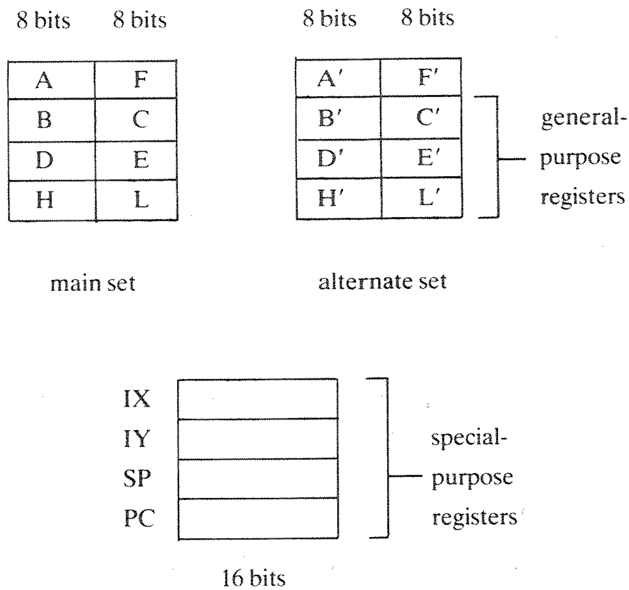
Third, we always have to handle 16-bit words, which is inconvenient if we're dealing with characters (which normally occupy a byte each). So it would be nice to allow 8-bit *and* 16-bit operations.

Fourth, the fact that there is only one general-purpose register (the A-register) can be annoying. It often means that intermediate results have

to be stored temporarily back in memory while some other calculation is done. The Z80 has a number of general-purpose registers; although, as we shall see, exactly how many there are varies depending on what we're using them for.

THE REGISTERS

Here's the register organization:



Ignore the alternate set for the moment. The registers appear in pairs, indicating that they may be used either as 8-bit or 16-bit registers. For instance, we can refer to the B-register (8 bits), or the C-register (8 bits) or the BC register (16 bits). The B, C, D, E, H and L registers can all be used in this way (in pairs BC, DE HL *only*) but the A and F registers are strictly 8-bit registers and cannot be combined. For the 16-bit pairs, the senior byte is the left-hand one (B, D, H) as you'd expect.

There are two index registers, the IX and IY, a stack pointer (SP) and program counter (PC). What, no indirection? Actually any of the 16-bit general-purpose register pairs (BC, DE or HL) can be used for indirection but, for simplicity, we shall always use the HL for this purpose.

There are two sets of instructions, one for handling 8-bit operations and the other for handling 16-bit operations. We'll start with the 8-bit "load" instructions.

How to shuttle data from one place to another...
and a few words about addressing modes.

17 Load

Let's look at the "load" (LD) operation as an example of the 8-bit group, It's very like the LD instruction in our imaginary machine, except that two extra addressing modes are allowed: *register-to-register*, and *immediate*. That gives a total of five, with *direct, indirect*, and *indexed* available as before.

1. *Direct addressing*
   This looks much the same as our imaginary equivalent, except that, since there is more than one register, we have to specify which register we want loaded:

   LA A,(3F1C)

   This loads the contents of 3F1C into the A-register. Note that, by convention, the movement is from right to left, so that we can write:

   LD (3F1C),A

   and mean "copy the contents of the A-register into 3F1C". (Actually, the A-register is the only *8-bit* register which can be directly addressed.)

2. *Indirect addressing*
   Again, this is straightforward. Since we're going to standardize on the HL for indirection, the instruction format it:

   LD A,(HL)

   which means "load the A-register *through* (i.e. from the address contained in) the HL register". To pass data in the opposite direction we could have:

   LD(HL),A

which puts the contents of A into the address *contained in* HL. (For this instruction, registers other than A are allowed.)

3. *Indexed addressing*
   Here, we need to indicate which index register is in use, and the amount of the offset:

   ```
   LD A,(IX + 2E)
   ```

   Note that in direct addressing, I showed an address of 4 hex digits, because 16 bits (2 bytes) are allowed for the address. The offset value in an indexed address instruction must be held in 1 byte, however, so I've only shown two hex digits.

4. *Register-to register*
   We can transfer data between registers like this:

   ```
   LD D,B
   ```

   which means: "load the contents of B into D".

5. *Immediate*
   Here, data itself, rather than the address of data, is placed in the address field. So we can write:

   ```
   LD B,07
   ```

   to mean "put the number 7 in B". Note again that the number is two hex digits, since it has to be stored in the single byte of the B-register. Note also that a "LD" is really a *copy*: the numbers are retained in their original addresses or registers, but a copy is placed at the destination.

HEX CODES

Now let's see what each of these instructions looks like in hex; for a full listing see Appendix 3.

1. LD A, (3F1C)

First we look up the opcode for the LD A, (nn) instruction. (The nn indicates a general 2 byte address). This is 3A. So you would expect the instruction to code as:

```
3A 3F1C
```

Unfortunately, there's a slight complication caused by the way the Z80 thinks about numbers; it likes the least significant (junior) byte of an address first. So we have to *swap the address bytes round:*

3A 1C3F

This is mildly annoying, but you soon get used to it. It is an invariable rule for 2-byte numbers in Z80 instructions: *junior byte first, then senior.*

The LD (nn), A instruction has the code 32, so:

LD (3F1C) becomes 32 1C 3F

2. LD A, (HL)

This is easy. There is no address part so it's just a 1-byte opcode. Look it up and you'll find it's 7E.

Similarly LD(HL),A codes as 77.

3. LD A, (IX + 2E)

The general instruction is LD A,(IX + d),d indicating a 1 byte displacement (in 2's complement notation), and its code is DD 7E. (Note that – a 2-byte opcode!) So the instruction is:

DD 7E 2E

where the byte 2E is the displacement chosen in this case.

4. LD D, B

No problem here, again. The code is 5Ø.

5. LD B, Ø7

The opcode is Ø6 so the instruction is Ø6 Ø7.

To kick off with, here's a machine code program to add two numbers—and a simple BASIC loader to write and run machine code easily.

18 Arithmetic

What about arithmetic? There's an ADD and a SUB instruction, both of which reference the A-register, and which may use any of the addressing modes except direct.

So let's try writing a program to add the numbers 4 and 7 together. This would work:

```
LD A,Ø4        [put 4 in the A-reg]
LD B,Ø7        [put 7 in the B-reg]
ADD A,B        [add them, and put the result in the A-reg.]
```

Now store the result away somewhere:

```
LD (7EØØ),A
```

Here's the program, the hex code, and the decimal equivalent:

Program	Hex	Decimal
LD A,Ø4	3E Ø4	62 Ø4
LD B,Ø7	Ø6 Ø7	Ø6 Ø7
ADD A,B	8Ø	128
LD (7EØØ),A	32 ØØ 7E	5Ø ØØ 126

LOADER

We're left with the problem of loading this code into the TRS-80, and then executing it. Since we're going to do a number of machine code

routines, it's going to be worthwhile writing a BASIC program which loads and then executes machine code.

But before we key in the basic we must ensure that it will coexist with our machine code routine. When you turn the machine on one of the things it asks for is "MEMORY SIZE?". Ordinarily we would just hit ENTER; but this time key in "32255" first and then hit ENTER. The machine will use 32255 as the highest address that BASIC can use, which will leave us with a 512 byte "attic" for machine language routines (32767 is the highest address of 16K RAM and 32767 − 512 = 32255).

If you've got a disk system, TRSDOS for example, the machine will pop you the "MEMORY SIZE?" question only after you've entered disk BASIC, at which point you can respond appropriately. Or, if you prefer, you can bypass the disk system entirely by holding down the SHIFT and the BREAK keys while you turn the machine on (or hit the reset key) and then proceed as above. Now for the loader.

This is fairly easy. In principle, all we need to do is to ask the user where he wants to put the code in memory, then ask for each byte of code in turn, and POKE it into the appropriate location. Then we run the program by calling the USR function. Finally, we PEEK all the program locations and data area to ensure that the program is still intact (remember, it's possible to overwrite a program by accident) and that the results are correct. Obviously, it makes sense to have the data and program areas adjoining. So we'll adopt this convention: the data area always precedes the program area, and is loaded with zeros to start with. So we'll begin by asking the user the size of his data area (as a number of bytes).

There's one other problem; according to the TRS-80 *Reference Manual* all routines called by USR have to end the same way.

```
RET   C9   201
```

We call this the *standard ending*. So we might as well make the program generate this code at the end of the routine automatically. Here's the loader in its simplest form.

```
10  PRINT "BASE ADDRESS: ";
20  INPUT B : PRINT B
30  PRINT "NO. OF DATA BYTES: ";
40  INPUT D : PRINT D
50  IF D=0 THEN GOTO 90
60  FOR I=0 TO D-1
70  POKE B+I,0
```

```
80  NEXT I
90  LET A=B+D
100 PRINT "CODE: ";
110 INPUT C
120 IF C<0 THEN GOTO 170
130 PRINT C
140 POKE A,C
150 LET A=A+1
160 GOTO 110
170 POKE A,201
```

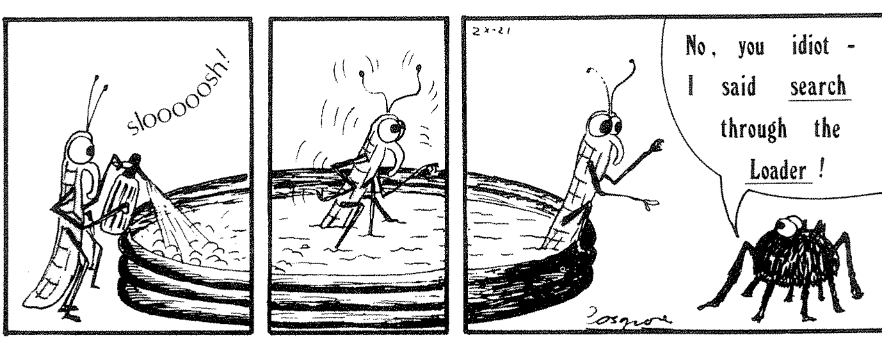
The TRS-80 manual says that USR functions fetch the starting addresses of their machine language routines at RAM 16526-16527. At this point our program's starting address (HEX 7E01) is stored in the variable A. So we'll add some code that posts the correct address for us. Note how the least significant byte of our machine language routine address goes into the first location (16526) and the most significant byte into the next.

```
180 LET Q = INT((B+D)/256)
190 LET R = (B+D)-256*Q
200 POKE 16526,R
210 POKE 16527,Q
```

Here's the USR function that calls our machine language routine.

```
220 LET Y = USR(20)
```

The value, here Y, returned by USR isn't usually needed, but it has to be there to satisfy the syntax of the statement. It actually contains whatever was in the BC register pair on returning from the machine code routine. The same is true for the argument 20. For our purposes any number will do.



Finally, we look at the state of the program and its data:

```
230 FOR I = B TO A+8
240 PRINT I,PEEK I
250 NEXT I
```

RUNNING

Now, to run the program. Load the "loader" program and run it. In response to its BASE ADDRESS request, type 32256, and, to NO. OF DATA BYTES, type 1. Finally, key in the machine code (62, 4, 6, 7 etc.) terminating with a negative value, a delimiter ignored on loading but signalling "end of code listing".

The system responds by printing the contents of bytes from 32256 onwards. In 32256 is 11, which is the sum of 4 and 7. This shouldn't surprise us much, since that's where we asked to store the result, and it's also the byte we allocated for data. The rest of the "memory dump" just confirms that the program is correctly stored.

Experiment, by altering the values being added. (Just POKE new values into 32258 and 32260 and GOTO 220). Or put the result somewhere else—32257, say. But to be on the safe side, reload the routine with an appropriate to the BASE ADDRESS or the NO. OF DATA BYTES before you try this one. In general, it's not a good idea to overwrite program code with data results! See how it changes the program?

In particular, try adding 240 to 100 (decimal). The result isn't 340! why? Think about it in binary:

```
240       1 1 1 1 0 0 0 0
100     + 0 1 1 0 0 1 0 0
  ┌───────0 1 0 1 0 1 0 0  =  84
  │       ───────────────
  ↓       1 1
  1
```

The sum generates a 1 in the ninth bit, which can't be held in an 8-bit byte, so it falls off the end and the quoted result is too small by the value of that ninth bit—256. No check has been made, no helpful error message printed. When you write machine code you're on your own. What you don't test for, you don't find out about.

AN IMPROVED LOADER

For that first try, I gave you a program that loaded *decimal* opcodes. That was so you could concentrate on the mechanics without worrying about hex codes. But hex is so much more convenient. Here's how to modify LOADER to accept hex, by combining it with the decimal/hex converter in Chapter 11. Beginning with line 11Ø through line 14Ø, change the code as follows:

```
11Ø  INPUT C$ : PRINT C$
114  IF C$ = "S" THEN GOTO 17Ø
118  LET DN = Ø
122  FOR I = 1 TO 2
126  LET H$ = MID$(C$,I,1) : LET E = ASC(H$)
13Ø  LET F = 48 : IF E > 64 THEN F = 55
134  DN = DN+16[(2-I)*(E-F)
138  NEXT I
14Ø  POKE A,DN
```

The procedure is exactly as before; but now at each input you key in the hex code: 3E, then Ø4, then Ø6, etc. Don't omit the zeros. Use "S" to end the inputs, in place of the previous "negative number" delimiter. That last POKE corresponds to the code POKE in the decimal loader.

From now on, I'm only going to give the hex codes. You can either modify LOADER as above, or you can convert from hex to decimal using Appendix 1. Since the latter is tedious, and it's easy to make errors, I strongly recommend the former.

There are 694 Z80 instructions: here's a selection of the more fundamental and accessible ones, and what they will do.

19 A Subset of Z80 instructions

I'm not going to describe every one of the 694 opcodes the Z80 has; that would be tedious and unnecessary. (But see Appendix 3.) We'll look at a subset of 3Ø-odd types of instruction (covering about 23Ø actual commands). Unfortunately, not all of them can use all the addressing modes. Here's a quick reference table showing which instructions can use what; the opcodes are given in Appendix 4.

Address Mode \ opcode	LD	ADD ADC SUB SBC AND OR XOR CP	INC DEC SLA SRA SRL	JR JRC JRNC JRZ JRNZ DJNZ	JP	JPZ JPNZ JPC JPNC JPP JPM	LD	ADD ADC SBC	INC DEC PUSH POP
Register	LD r, s	ADD A, r	INC r					ADD HL, r	INC r
Immediate	LD r, n	ADD A, n			JP nn	JPZ, nn	LD r, nn		
Direct	LDA, (nn) LD (nn), A						LD HL, (nn) LD (nn), HL		
Indirect	LD A, (HL) LD (HL), A	ADD A, (HL)	INC (HL)		JP (HL)				
Indexed	LD A, (IY + d) LD (IY + d), A	ADD A, (IY + d)	INC (IY + d)	JR d					
	8-bit operations				16-bit operations				

The notation in the table needs some explanation. Some of the opcodes will be unfamiliar, but we'll deal with those later. Otherwise, the conventions are:

1. Each entry in the table shows an example of the format of the instruction type. Any of the other opcodes in that column could be substituted.

2. "r" or "s" denotes any register. Whether this is an 8-bit or a 16-bit register depends on which part of the table the instruction is in. For instance, in the LD r, s instruction, r and s are any 8-bit registers (A, B, C, D, E, H, or L), but in ADD HL,r "r" is one of BC, DE, HL, SP.
3. "n" is any 8-bit number. "nn" is any 16-bit number.
4. If a register is explicitly stated, as in LD A, (nn), then this is the only register which may be used for this purpose.
   This is a wild oversimplification. Sometimes, other registers are usable, but the point is that the set of instructions I've shown are always OK and you can worry about extending your vocabulary of instructions when you're handling this lot confidently.
5. "d" is any 8-bit number, but it's always added to some 16-bit value. In other words, it's an indexing displacement.

Now let's look at the new opcodes:

AND

This operation takes the contents of the A-register, and another 8-bit field, and examines these, bit by bit. Only if corresponding bits are both "1" does it put a "1" back in this position in the A-register. Otherwise it inserts a "Ø".

For instance, AND A, Ø7 has the following effect:

A-register before the operation:	Ø Ø 1 1 Ø 1 Ø 1
Ø7:	Ø Ø Ø Ø Ø 1 1 1
A-register after the AND:	Ø Ø Ø Ø Ø 1 Ø 1

See how the junior three bits have been transmitted? So you can use AND to select a portion of a byte.

OR

This works in a similar way to AND, but this time, the resulting bit is a "1" if either of the initial bits is a "1". So OR A, Ø5 gives

A-register before:	Ø 1 Ø Ø 1 Ø 1 1
Ø5:	Ø Ø Ø Ø Ø 1 Ø 1
A-register after:	Ø 1 Ø Ø 1 1 1 1

Now, certain bits are being forced to "1" regardless of their original value.

XOR

Here the initial bit values must be different for the result to be a "1", XORA A, B3 gives:

A-register before:	Ø 1 Ø 1 1 Ø 1 Ø
B3:	1 Ø 1 1 Ø Ø 1 1
A-register after:	1 1 1 Ø 1 Ø Ø 1

It's particularly useful for flipping a register from Ø to 1 and back again. If the A-register contains Ø to start with, every time the instruction XOR A,Ø1 is executed, the value in the A-register will flip. (Ø to 1, back to Ø, back to 1 and so on.)

CP

This is the "Compare" instruction. The contents of the A-register are compared with those of another 8-bit field. That raises a problem, though: how is the result of the comparison signalled?

This is what the F (or *flags*) register is used for. Each bit of the F-register holds some information about the effect of the last instruction to alter them. (Not all instructions do alter them).

The flags which most interest us are the Carry, Zero, Overflow and Sign flags. CP can alter any of these, but the one of most significance here is the Zero flag, which is set if the two values being compared are equal.

If the A-register contents are *less* than those of the compared byte, the sign flag is set. This is equivalent to saying "the result is negative". This is all you need to know about the flags at the moment; it's an intricate topic if you delve deeper.

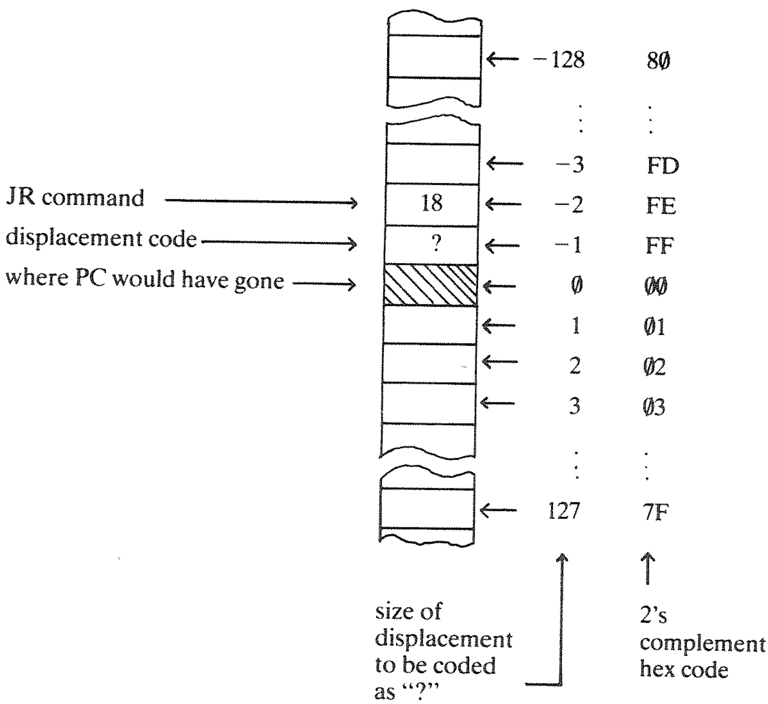
THE JUMPS

All the conditional jumps branch (or not) depending on the contents of the flags. So, for instance, JPZ says "jump if the Zero flag is set". Now we can see how the CP instruction can be used. Suppose, for example, that we wish to see if a particular byte, pointed at by HL, contains 1E hex. If it does, we want to branch to 7E32. The code is:

```
LD A, 1E        3E 1E
CP A,(HL)       BE
JPZ 7E32        CA 32 7E
```

All the other jumps behave similar; JPNZ says "jump on a non-zero result" (zero flag *not* set), JPP says "jump on a positive result" (sign flag *not* set), and so on. All of them have one thing in common, and that is that the address of the jump is fixed. In other words, if, for any reason, we would like a routine to run somewhere in memory other than where we first loaded it, all the jump addresses must be changed. The Z80 deals with this neatly by allowing "relative jumps" (JR). In other words, you can jump so many butes forward (or back) from where you are. This displacement is held (in 2's complement notation, Appendix 1) in 1 byte, so the distance which can be jumped can't exceed 128 bytes backwards or 127 bytes forwards.

The displacement is calculated from what the PC value *would* have gone to next, had no jump occurred; namely, the address of the next command in the program. Like this:



Here's an example. We want to examine each byte of memory in turn for the first occurrence of 1E hex. Assume for simplicity that the start address is already in HL. We could write:

```
        LD A, 1E
LOOP:   CP A, (HL)
        INC HL
        JRNZ LOOP
```

Two points need explaining. First, I've sneaked in a new instruction: INC. This is short for INCrement. It just adds 1 to the contents of the specified register; so the compare operation is always looking at the next memory byte because HL is being bumped up by 1 every loop. (By the way, DEC, short for DECrement, does exactly the opposite.) The second point is that there's no obvious difference between JRNZ LOOP and JPNZ LOOP. It isn't until we *assemble* the instructions into machine code that the difference is clear. Suppose the code is to be loaded from 43ØØ hex:

Address		Instruction	Hex code
43ØØ		LD A, 1E	3E 1E
43Ø2	LOOP:	CP A, (HL)	BE
43Ø3		INC HL	23
43Ø4		JRNZ LOOP	2Ø FC

Why is FC in the address part of the JRNZ instruction? It works like this: when the JRNZ instruction is executed the PC is bumped up by 2 because it's a 2-byte instruction. So the PC is now at 7EØ6. We want to jump to LOOP, which is at 7EØ2, 4 bytes back, or −4 bytes away, to use the Z80's way of thinking about it. Now, 4 in binary is ØØØØØ1ØØ and we create −4 by flipping the bits and adding 1 (2's complement, remember?). So:

```
Ø Ø Ø Ø Ø 1 Ø Ø
                      flip the bits
1 1 1 1 1 Ø 1 1
            + 1       add 1
---------------
1 1 1 1 1 1 Ø Ø
                      convert to hex
   F       C
```

Another thing which may be worrying you: INC HL doesn't alter the flags, so I'm safe to test after the increment.

The same program with absolute jumps would have looked like:

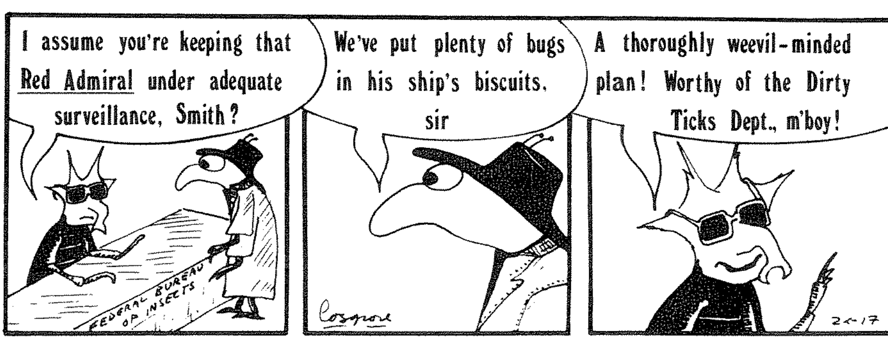
Address		Instruction	Hex code
43ØØ		LD A, 1E	3E 1E
43Ø2	LOOP:	CP A, (HL)	BE
43Ø3		INC HL	23
43Ø4		JPNZ LOOP	C2 Ø2 43

Notice that the JPNZ instruction has 3 bytes because it contains a whole 16-bit address; and don't forget about swapping the 2 bytes of that address around!

There's one very powerful instruction in the Jump group I haven't mentioned yet—DJNZ. It decrements the B-register by 1 and jumps (relative) only if the result is non-zero.

Suppose our little "search for 1E" program is only to search a region one hundred (hex 64) bytes long, after which it should leave the loop whether it's found a 1E or not:

	LD B, 64	Ø6 4Ø
	LD A, 1E	3E 1E
LOOP:	CP A, (HL)	BE
	JPZ GOTCHA	CA — — (address for GOTCHA)
	INC HL	23
	DJNZ LOOP	1Ø F9



The loop is executed one hundred times, unless a 1E is found, in which case a branch to GOTCHA occurs. In other words, DJNZ acts like a simple FOR loop in BASIC.

Note that with *all* the relative jump commands JR, JRC, JRNC, JRNZ, and JRZ, the size of jump is calculated the same way. A table of 2's complement hex codes is given in Appendix 1 for hand-coding of jumps.

ADC and SBC

These are the "ADD with Carry" and "SUB with Carry" instructions. I said earlier that there is a Carry flag in the flags register. This gets set if there is a carry generated out of a register by an arithmetic instruction. The ADC instruction will act just like ADD, except that it will add 1 more in if the Carry bit has been set by a previous operation. The SBC instruction works the same way, except that it will subtract the Carry flag.

THE SHIFTS

The shift instructions, SLA, SRA and SRL, all have the effect of shifting bit-patterns around.

SLA shifts the pattern left by 1 bit, so if the B register contains:

Ø	Ø	1	Ø	1	1	Ø	Ø

and SLA B is executed, the result is:

Ø	1	Ø	1	1	Ø	Ø	Ø

(Notice that a zero is used to fill on the right.)

Since ØØ1Ø11ØØ = 44 and Ø1Ø11ØØØ = 88 (decimal) you can see that the effect is to multiply by 2.

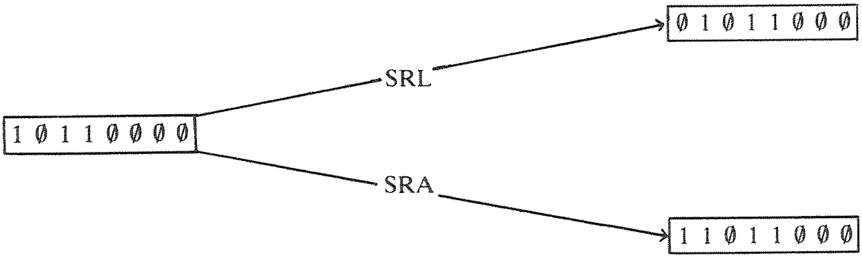
Another SLA B will give:

1	Ø	1	1	Ø	Ø	Ø	Ø

Since the senior bit is now 1, this will be seen as a negative number, and the Sign flag will be set. So far as the programmer is concerned, what's happened is that the value (176) can't be held in a byte, so we've got an overflow condition.

Right shifts work much the same way, but there's one important thing to note: SRL fills the senior bit with a zero, but SRA fills with whatever was there before.

For instance:



The reason is this: SRL is a *shift right logical*, which simply shifts the bit pattern without alterning it. SRA is a *shift right arithmetic*, which treats the operation as "divide by 2." Now, when a negative number is divided by 2 the result should still be negative, so we have to preserve the sign bit.

PUSH and POP

You'll probably remember these terms from our discussion on stacks. They're used here in exactly the same way, and allow us to access the machine stack other than through a subroutine call.

This can be useful for saving values temporarily. For instance, suppose you've got a value in BC which you want later, but just now you'd like to use BC for something else. You can write:

```
PUSH BC
..........
Code
using
BC
..........
POP BC
```

This is often done before a subroutine CALL as well, so that it doesn't matter what registers the subroutine uses: it can't interfere with the calling program's data. You may see code like:

```
PUSH BC
PUSH DE      save the registers
PUSH HL
CALL 4FA1
POP HL
POP DE       restore register values
POP BC       (note the order!)
```

assuming that the A-register is manipulated by the routine, so we don't need to save it.

Warning

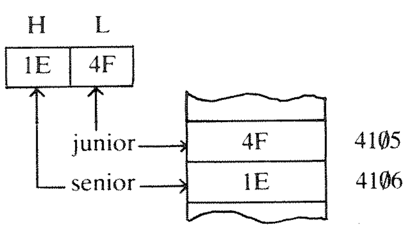
Unless you deliberately choose to alter it, the stack pointer SP will be set according to the operating system of the TRS-80. There's no harm in leaving it at that value, provided you make sure that PUSHes and POPs cancel out in pairs, so that SP returns to its initial value on leaving the machine code routine. Similarly CALLs and RETs have to match. (USR generates a CALL, matched by the final RET that is tacked on to the end by the LOADER routine.)

A 16-BIT QUIRK

One feature of the 16-bit operations (PUSH, POP, LD in particular) which is important to grasp is the order in which bytes are transferred from register to memory and vice versa. It's like this:

```
LD (41Ø5),HL
```

will have the following effect, if HL contain 1E4F:



In other words, the least significant or "junior" byte in the register is loaded into the specified address; and the most significant or "senior" byte is loaded into the by following this. Conversely,

```
LD HL,(41Ø5)
```

would have exactly the reverse effect. (NB: it codes as 2A Ø541, following the standard convention!) Similarly

```
LD HL 1ØØØ
```

(an attempt to load HL with the value 1ØØØ hex) encodes as

```
21 ØØ 1Ø
```

so that, even though 1ØØØ is data, not an address, its bytes get transposed as usual.

CRASHES

When a BASIC program crashes, there's little harm done: you can always break out, one way or another, without losing the program. But machine code crashes are more spectacular, and infuriating. Spectacular, because they often signal their presence by drawing op-art patterns all over the screen; and infuriating because the only way to break out of them is to pull the power plug out or hit the reset button, either way you lose the contents of RAM. You want to see a crash, OK, try this little variation of a program in hand. Remember the hexmas tree program back on page 109, the one with all the DATA statements ? Replace all of them with:

```
1Ø  DATA 2Ø5,2Ø1,ØØ
```

Adjust all the line numbers and modify the loop so that it only READs and POKEs the three DATA items. Now run it.

The machine returns to BASIC. Everything seems all right, but wait a minute. Try listing your BASIC program. What happened to it?

Our machine language routine, via a subroutine call, loaded the address ØØ2Ø1 into the program counter. Now, ØØ2Ø1 is a ROM address and, more specifically, is a part of the machine start up routine so the machine was trying to turn itself on but because we jumped in the middle (it begins at ØØØØØ) much of the routine was skipped. Notice that the computer did not ask you for the cassete rate or the memory size.

As crashes go this one isn't so bad. Once a crash occurs, you're stuck with it: pull the plug and start again. (However, there's no way to alter the ROM contents, so don't worry about doing any lasting harm! It is you, not the TRS-80, that will suffer!) But there are some simple precautions worth taking.

1. Check all machine code listings scrupulously and make sure you have input them correctly.
2. *Never* use HALT (hex code 76).
3. Make sure that CALLs and RETs match, as do PUSHes and POPs.
4. Make sure you call the correct starting address.
5. Unless there's not much to lose. SAVE what you can on tape or disk before calling USR.

Machine code has no instruction for multiplying numbers; but you can do it if you combine arithmetic, logic, and shifts. Digging deeper now...

20 A Machine Code Multiplier

Now let's write a few simple routines. Remember I said that there's no Z80 multiply instruction? Let's write a subroutine to do the job.

AN EXAMPLE

First we should examine the nature of the problem, and there's no better way of doing that then looking at an example. We'll keep things a simple as possible, and work in 8-bit registers; so if we want to multiply 9 by 13 it will look like:

```
  00001001
× 00001101
```

Now we can treat this as a conventional long multiplication, but because it's in binary, it's actually easier than usual; if the current digit we're multiplying by is 1, copy the top line; if it's zero, do nothing:

```
  00001001     P
× 00001101     Q
  --------
  00001001
  00100100
  01001000
  --------
  01110101
```

Of course we've had to add in zeros on the right at each stage, just as we would in a decimal long multiplication. In machine code terms, that's equivalent to a shift left. I've called the two numbers P and Q, for reference.

While P is shifted left, it's also going to be convenient to shift Q right, because that way we only need to keep examining the junior bit of Q to determine whether to add P into the sum or not.

PROCEDURE

Assume that P and Q are in the D and E registers. The procedure is:

1. Set the A-register to zero.
2. If the junior bit of E is 1 then add D into A.
3. Shift D left.
4. Shift E right.

repeat these steps 8 times (steps 2–4)

Here's a first stab at the code:

```
LD A,00
LD B,08
```

The first step's obvious; the second sets B to act as a loop counter in conjunction with a DJNZ to come at the end. Now we want to test the junior bit of E. The only way we currently have of doing that is to use a mask pattern (00000001) with an AND operation, so let's set up the C register to that pattern:

```
LD C,01
```

We can only AND with the A-register, which will destroy its current contents, so we'll save it in L first:

```
LOOP:   LD L,A
```

then extract the junior bit of E, and restore the A-register:

```
LD A,C
AND A,E
LD A,L
```

If the result of the AND was zero, we need to jump round the "add D into A" part of step 2 so:

```
JRZ SHIFT
```

(Note that since LD doesn't affect the flags, the JRZ still refers to the AND.) Otherwise perform the ADD:

```
ADD A,D
```

Now do the shifts:

```
SHIFT:  SLA D
        SRA E
```

and see if we've done the loop enough times yet:

```
        DJNZ LOOP
        RET
```

THE CODE

Here's the whole thing:

Address		Instruction	Hex code
ØØØØ		LD A, ØØ	3E ØØ
ØØØ2		LD B, Ø8	Ø6 Ø8
ØØØ4		LD C, Ø1	ØE Ø1
ØØØ6	LOOP:	LD L, A	6F
ØØØ7		LD A, C	79
ØØØ8		AND A, E	A3
ØØØ9		LD A, L	7D
ØØØA		JRZ SHIFT	28 Ø1
ØØØC		ADD A, D	82
ØØØD	SHIFT:	SLA D	CB 22
ØØØF		SRA E	CB 2B
ØØ11		DJNZ LOOP	1Ø F3
ØØ13		RET	C9

If you want to try this program out, you'll have to arrange for the D and E registers to hold the values to be multiplied. So you could precede the program by something like:

```
LD HL,7EØØ     21 ØØ 7E
LD D,(HL)      56
INC HL         23
LD E,(HL)      5E
```

and then POKE 7E00 (hex) and 7E01 (hex) with the values to be multiplied, before calling the program. These two bytes will, of course, be the two zero bytes at the beginning of the routine, so the LD HL, 7E00 will start in 7E02. Note that I didn't assign actual addresses to the program, but simply started at zero. This is because all the jumps are relative, so actual addresses are unimportant; only displacements matter. For example, with 16K you can replace all 43s in the above by 7Fs, to work with a 256-byte attic.

You'll also need to *output* the answer: at the moment it's just sitting in the A-register. A simple way to do this is to stick the answer into the display file (see next chapter for details) by adding the following code at the end, *in place of* the C9 (RET) instruction, which is there only because I said this was going to be a *sub*routine.

```
0013    LD HL (3C 00)    21 00 3C
0016    INC HL           23
0017    LD (HL) A        77
```

Add this at the end; add the bytes 07 and 08 at the front (or POKE them later); enter using LOADER with 2 data bytes. The number "7" will appear at the top corner of the screen. The code for 7 is 56; and that's the product of the two numbers 07 and 08 you POKEd in. Of course a more elegant display routine would be nice: think about PRINT USR or read the next chapter and design one. But for a test, this method suffices.

BIT

Now, I have a confession to make; there's an easier way of testing to see if the junior bit of E contains 1. There's an instruction BIT0, E which does the job. So:

```
LOOP:  LD L,A     6F
       LD A,C     79
       AND A,E    A3
```

becomes just:

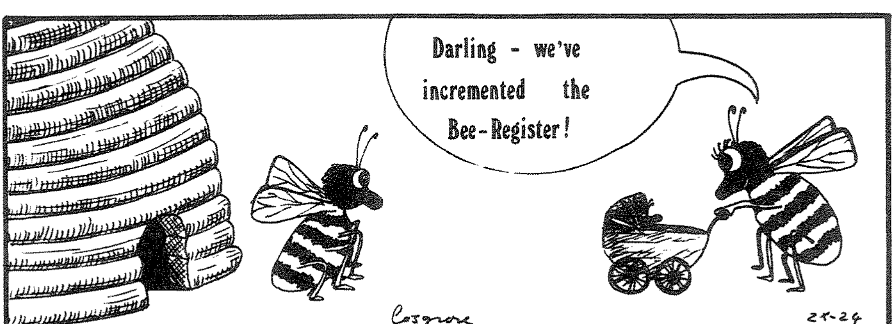
```
LOOP:  BIT0,E     CB43
```

and the LD A, L has to disappear as well.

Why didn't I tell you that in the first place? Well, firstly, I promised

to use only the subset of instructions in the table, a promise I've now broken. But I've made an important point in the process—that it's possible to do things satisfactorily without knowing the full instruction set.

This has been something of an academic example; I chose it because it uses several common instructions in conventional, but not necessarily obvious, ways, but I'm not suggesting that you will find a need for dozens of 8-bit integer multiplications.



To control graphics from machine code, you must know where the display is stored, and how to alter it. Change the whole screen to inverse video, or draw a line of characters, all in a flash!

21 The Display File

So now for something more clearly useful. Fed up with BASIC's ability to draw graphics lines at a snail's pace? Let's see if we can't write a machine code subroutine which will draw straight lines from any point on the screen running horizontally, vertically, or diagonally.

At least, that's the target. Let's deal with the problem in easy stages. Obviously, we need to know something about how the TRS-80 handles displays before we can get anything on the screen. As you've probably seen from the *Manual*, there's an area of memory called the *memory mapped video display* from which the screen display is generated. All we have to do is store character values in this region to get them displayed.

The file is a 1Ø24 byte section of RAM beginning at address 1536Ø. The first 64 bytes of the file form the top row of the screen. The second 64 form the second row and so on until you have 16 rows, which fills the screen. This 16 by 64 approach is no accident as 16 and 64 are both powers of two which ties in nicely with our binary computer system.

DISPLAYING A CHARACTER

Anyway let's try something simple, like putting a graphics symbol at the top left-hand corner of the screen. If the screen is blank to start with, there will be just 16 new lines in the display file. So our problem is to overwrite the second of them (where the "A" is in the above example) with our chosen symbol.

First we load HL with an address within the display file, 158ØØ for instance:

```
LD HL,3E9Ø          21 9Ø 3E
```

Now put the graphics character (88 hex, say) in the A-register:

```
LD A,88            3E 88
```

and finally, put this value where HL is pointing:

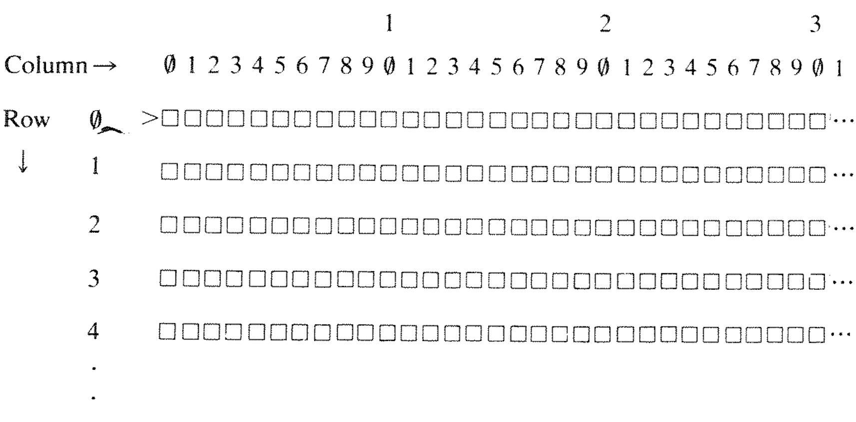
```
LD (HL),A          77
```

LINE-DRAWING

To get a horizontal line, 1Ø characters long, on the top line of the display we could execute the following code:

```
        LD A,88       3E 88      set value to be displayed
        LD B,ØA       Ø6 ØA      set loop count
        LD HL,3CØØ    21 ØØ 3C   point to first character in display file
LOOP:   LD (HL),A     77         display
        INC HL        23         point to next character
        DJNZ LOOP     1Ø FC      do it again
```

To do the same job anywhere else on the display, all we need to do is alter the value in HL to start with by an appropriate offset. In principle it's easy to calculate the necessary offset. Let's think about the display file like this:



If the HL is loaded with 36ØØ so that it points at column Ø, row Ø, then we simply multiply the row number we want by 64 and add on the column number. That is:

offset = row*64 + column

Provided the row value never exceeds 7, we could use our 8-bit multiplier here. But there's a neater way:

```
offset  =  row*(32 + 1) + column
           row*32 + row + column
```

Despite the fact that this expression for the offset seems more complicated than the original, it has the advantage that the multiplication is now by a power of 2 (2 raised to the fifth power), so all we have to do is shift row left 5 times to evaluate row * 32.

Now let's imagine that the row value is available in the E-register, and the column value is in the C-register. We can calculate the offset like this:

```
        LD B,Ø5         Ø6 Ø6
SHIFT:  SLA E           CB 23
        DJNZ SHIFT      1Ø FC
```

Hold on, it's not quite as easy as that! This piece of code shifts the E register contents left 6 times all right, and that's fine if row * 64 is less than 255, but it could easily be more than that, and the E-register will overflow. So we need a 16-bit register. If we use DE, the above code can be used as a basis for the routine, but there are some pieces to add on. First, we'll have to make sure that D contains zero to begin with. Second, as bits shift left off the end of E we want them to appear in D and then shift along D. This will work:

```
        LD D,ØØ         16 ØØ    clear D
        LD B,Ø5         Ø6 Ø6    load loop count into B
SHIFT:  SLA D           CB 22
                                 shift left DE
        SLA E           CB 23
        JRNC EOL        3Ø Ø1    go to End of loop on no carry
        INC D           14       put the carry into the junior bit of D
EOL:    DJNZ SHIFT      1Ø F7    test for end of loop
```

Now we want to add this into HL, having first loaded it with the first address of the display file:

```
        LD HL,(3CØØ)    21 ØØ3C
        INC HL          23
        ADD HL,DE       19
        ADD HL,BC       Ø9
```

Now we simply execute the "draw a line" routine as before:

```
        LD A,88          3E 88     (or whatever)
        LD B,ØA          Ø6 ØA
LOOP:   LD (HL),A        77
        INC HL           23
        DJNZ LOOP        1ØFC
```

The hex codes are given below, tidied up.

This routine produces a horizontal line because of the INC HL instruction in the loop. Change HL be some value other than 1, and we get different shapes. INC HL twice, and every other print position will display the character, for instance. Add 64 (decimal) into HL in every loop and we get a vertical line. Add 65 (decimal) into HL in each loop and we get a diagonal line.

You could have a library of such routines and simply call one whenever you want that kind of line.

LISTING

Here's the complete code. This time we won't bother with addresses in the listing: they're not important (thanks, once again, to *relative* jumps).

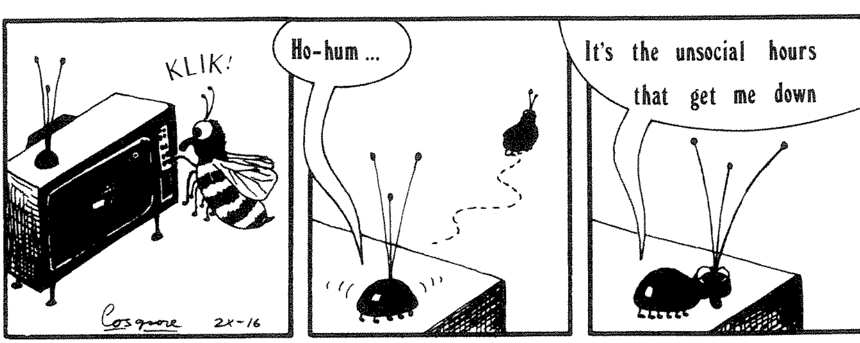
```
        LD C,ØØ          ØE ØØ
        LD E,ØØ          1E ØØ
        LD HL,(4ØØC)     2A ØC 4Ø
        INC HL           23
        LD D,ØØ          16 ØØ
        ADD HL,DE        19
        LD B,Ø5          Ø6 Ø5
SHIFT:  SLA D            CB 22
        SLA E            CB 23
        JRNC EOL         3Ø Ø1
        INC D            14
EOL:    DJNZ SHIFT       1Ø F7
        ADD HL,DE        19
        ADD HL,BC        Ø9
        LD B,ØØ          Ø6 ØØ
        LD A,ØØ          3E ØØ
```

```
LOOP:  LD (HL), A       77
       LD DE, ØØ ØØ     11 ØØ ØØ
       ADD HL, DE       19
       DJNZ LOOP        1Ø F9
```

The zero bytes underlined must be POKEd before calling the routine, as follows:

Start address + 1:	starting column (e.g. Ø5 for column 5)
Start address + 3:	starting row (e.g. Ø7 for row 7)
Start address + 25:	number of characters to be plotted (e.g. ØA)
Start address + 27:	code of graphics character
Start address + 3Ø:	value added to HL between plots (e.g. Ø1 for a horizontal line, 21 for a vertical line, 2Ø or 22 for diagonal lines)
Start address + 31:	not normally used unless the value to be added exceeds 255, otherwise set to ØØ

Once you've loaded this up, and seen what it does, think about incorporating it into BASIC programs to generate, say, a series of squares. Use RND to find the top left-hand corner (column and row) and the length of side. Then POKE the relevant addresses in the machine code routine, and call it via USR. Do this four times for the four sides of the (open) rectangle. Don't forget to test the sizes to see if it will all fit on the screen!



Some new, powerful commands: block search and block transfer. The alternate registers. And programs to SCROLL limited parts of the screen, or scroll sideways at high speed.

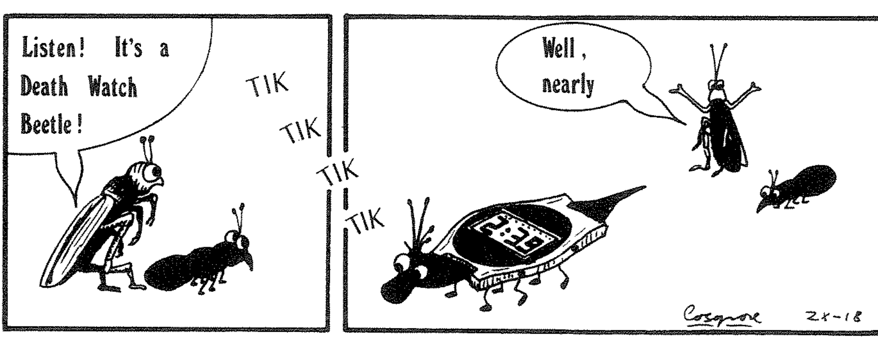
22 Some Things I Haven't Told You

I've deliberately simplified (even oversimplified) in places in this chapter, and I don't appologize for that. After all machine code isn't the easiest thing to tackle, and listing all its features at once is just confusing. On the other hand, if you looked at other books on Z80 machine code, you may well be wondering why I have frequently done things in a roundabout way.

So I'll try to redress the balance now.

BLOCK SEARCH

First, there are some very powerful instructions which will search a whole block of memory. I'll take CPDR which is short for "compare, decrement and repeat" as an example.



If I write a piece of code like this:

```
        LD BC, Ø1ØØ     Ø1 ØØ Ø1
        LD HL, 5ØØØ     21 ØØ 5Ø
        LD A, Ø5        3E Ø5
        CPDR            ED B9
NEXT:   ——
```

what happens is this:

When the CPDR instruction is encountered the value in the A-register is compared with the contents of the byte at which HL is pointing. If they are equal, control passes to NEXT: If not, BC and HL are both decremented by one, and the "compare" is repeated until a match is found or until BC contains zero. In other words, those four instructions say: "Find the first occurrence of a byte containing Ø5 from address 5ØØØ (hex) down to address 4FØØ and leave HL pointing at it. If there isn't one, set BC to zero."

So the first example I gave of using jumps, which was a little "compare" loop, could have been done much more easily. But, of course, it wouldn't have illustrated jumps!

BLOCK TRANSFER AND PARTIAL SCROLLING

Secondly there are some *block transfer* commands LDIR and LDDR which are invaluable for shifting blocks of data around in memory. For instance, to use LDIR, you:

- Load HL with the address of the first byte to be transferred.
- Load DE with the address of the first destination byte.
- Load BC with the number of bytes to be moved.

Then LDIR transfers the first byte; increments HL and DE; decrements BC, and keeps doing this until BC hits zero. LDDR is similar, but it decrements HL and DE (and decrements BC as before). To see LDIR in action, here's a very useful routine that lets you SCROLL a band of columns on the screen, leaving the rest fixed.

```
        COLUMN            07 00        set up
        WIDTH             09 00        data
        LD A,00           3E 00
        LD HL, 3C00       21 0036      get ready
        LD DE, (COLUMN)   ED 5B 00     for block
        ADD HL, DE        19           transfer
        INC HL            23
LOOP:   LD D, H           54
        LD E, L           5D
        LD BC, 0040       01 40 00     shift section
        ADD HL, BC        09           of a line up
        PUSH HL           E5           to line
        LD BC, (WIDTH)    ED,4B 02 7E  above
        LDIR              ED B0
        POP HL            E1           test and
        INC A             3C           repeat until
        CPA, 16           FE 0_        last line
        JRNZ, LOOP        20 ED        reached
```

The numbers in COLUMN and WIDTH can be POKEd to change them: here I've set up a scroll starting in column 7 and of width 9, which means that columns 7,8,9,10,11,12,13,14,15 will scroll and the rest will not. There are 4 data bytes before the start address.

To see this in action, you'll need to give it something *to* scroll: a clear screen isn't too dramatic! One way to do this is to add a few lines to LOADER:

```
212  FOR I = 1 TO 8
214  PRINT "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
215  PRINT "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB"
216  NEXT I
```

And, for scrolling more than one line, put the USR instruction inside a loop:

```
218  FOR I = 1 TO 13
220  LET Y = USR (20)
222  NEXT I
```

Experiment, putting different designs on the screen before scrolling.

The underlined 10 can be changed (but only to something smaller)

to scroll only down to a certain row. By changing the start, you can select a rectangle and scroll that. But, you'll need to think about what happens to thee bottom line, or the scroll will just copy it out again in the same place.

You *can* write a BASIC partial scroll but it is much too slow to be interesting.

SIDEWAYS SCROLL

Here's another useful routine in the same vein. It scrolls the whole screen sideways, as if the display were cylindrical. In each line of the display the first character is pushed on to the stack, the rest are shifted down, and then the first one is popped off again to the top end. This is repeated for all lines of the display.

```
        LD A, Ø             3E ØØ        set loop count
        LD HL, 3CØØ         21 ØØ3C      initialize
        LD D, H             54           for block
        LD E, L             5D           transfer of
        INC HL              23           first line
LOOP:   PUSH AF             F5           save loop count
        LD A, (DE)          1A
        PUSH AF             F5
        LD BC, 63 dec       Ø1 3F ØØ     rotate one
        LDIR                ED BØ        row one
        POP AF              F1           left
        DEC HL              2B
        LD (HL), A          77
        POP AF              F1increment
        INC A               3C           loop count
        INC HL 2 TIMES      23 23        adjust for
        INC DE              13 13        next row
        CP A, 16dec         FE 1Ø        jump unless
                                         loop count is on
        JRNZ LOOP           2Ø EC        last row
```

To see this at its best, embed it in a BASIC loop so the screen scrolls repeatedly; and give it a nice display to scroll. For instance:

```
212  FOR I = Ø TO 15
214  PRINT AT I,I; "SCROLL"
```

```
216  NEXT I
22Ø  LET Y = USR (2Ø)
222  GOTO 22Ø
```

MNEMONICS

The next thing I should mention is that some people use slightly different mnemonic opcodes from those I've described. For instance, where I would write LDA (nn), some people write LD (nn). This is because the A-register is the only register which can be loaded directly, so it's not strictly necessary to specify it. I find that actually quoting it every time is a useful aid to memory, though.

ALTERNATE REGISTERS

I said that there is an alternate set of registers, and then promptly ignored them. You can always get away without using them, and they aren't very useful anyway. You can't do any arithmetic in them. Their main use is to save temporarily the contents of the main set while you're executing some routine which alters the main register contents in ways you don't want. This is done by exchanging the contents of the main and alternate sets before, and again after, the offending routine:

```
EX AF, AF'    Ø8       swap AF with AF'
EXX           D9       swap BC, DE, HL with BC', DE', HL'
CALL . . .    CD — —   call offending routine
EX AF, AF'    Ø8       restore registers
EXX           D9
```

Of course, you could do the same thing by PUSHing the register contents you want saved onto the stack before the CALL, and POPing them off afterwords.

FLAGS

I steered clear of the technicalities of the F-register; but here's a brief summary: for more details consult one of the machine code books in the bibliography.

There is room in the F-register for eight flags; but it only uses six of its available bits. The flags are:

C	Carry flag
Z	Zero flag
S	Sign flag
P/V	Parity/Overflow flag
H	Half-Carry flag
N	Subtract flag

These are arranged in the register like this:

S	Z	X	H	X	P/V	N	C

where X means "not used."

The Carry flag is affected mainly by add, subtract, rotate and shift commands, and we've already seen how it works.

The Zero flag is affected by a tremendous number of commands. Roughly, if anything (except LD, INC, DEC) changes the contents of A, then the Zero flag is set (to 1) if A is zero, and reset (to Ø) otherwise. BIT sets the flag if the specified bit is zero. CP sets or resets the flag according to the result of a comparison.

The Sign flag stores the sign bit of the result of whichever operation has just been carried out: 1 for negative, Ø for positive.

The P/V flag works differently on arithmetic or logical commands. In arithmetic, it is set if there is an *overflow* in 2's complement arithmetic (e.g. if the sum of two positive numbers runs off the end of the accumulator, giving an apparently negative result). For a logical operation it is set to Ø if the byte in A has an *even* number of bits equal to 1; and to 1 if the number of bits equal to 1 is *odd*. The odd/even character of the number of bits equal to 1 is called the *parity* ofthe byte.

The H and N flags are used only for binary coded decimal calculations, and can be ignored.

ROM ROUTINES

I have also been guilty of reinventing the wheel now and then. The point is that the BASIC interpreter in ROM has to call on routines such as those we've developed. So why not simply *call* them, rather than write our own? In general, the answer is that it would have been much more sensible to do so, since it saves a lot of effort and, almost as important, computer memory. But—so far as *this* book is concerned, my aim has been to tell you about Z80 machine code, and avoid, as far as possible,

the special features of the TRS-80. If all the examples had consisted of a series of calls to addresses in ROM you wouldn't have learned much! Of course, to use the ROM routines, you need to know where they are.

EFFICIENT USE OF MACHINE CODE

Finally, I want to tie up two loose ends I left hanging right at the beginning. I said that there may be reasons other than BASIC's need to interpret statements each time they are executed for a machine code program to run faster than BASIC.

I'll explain with an example:

BASIC		Machine code
10 FOR I = 20 TO 1 STEP -1		LD B, 14
....	LOOP:	....
50 NEXT I		DJNZ LOOP

In each case, every time the loop is executed a variable is decremented by 1. But that process is much more complicated in BASIC than it is in machine code. The reason is that since BASIC has to deal with decimal values some of the time, it assumes that it's doing so all the time, and so it actually subtracts 1.00000000, which is no easier than subtracting 1.58712684. In fact, the procedure employed is quite complex and time-consuming. The machine code, on the other hand, uses a single, purpose-built instruction. The result is in the region of 100 times faster.

The other point I deferred was that machine code can occupy more memory than its BASIC equivalent. Here's an example which illustrates why:

BASIC	Machine code	No. of bytes
30 IF R = P AND P = Q THEN		
LET P = W	LD HL, 5000	3
	LD A, (HL)	1
	LD HL, 5001	3
	SUB A, (HL)	1
	JRNZ NEXTBIT	2
	LD HL, 5001	3
	LD A, (HL)	1

```
            LD HL, 5002       3
            SUB A, (HL)       1
            JRNZ NEXTBIT      2
            LD HL, 5001       3
            LD A, (5003)      3
            LD (HL), A        1
NEXT BIT:                    27  TOTAL
```

The machine code assumes that R, P, Q, and W are held in the bytes 5000, 5001, 5002 and 5003, respectively. In practice it wouldn't be as simple as that because each number will occupy 5 bytes and the SUB will actually be a CALL to a floating point subtraction routine. In any event, the actual code would need at least as many, and probably more than, the 27 bytes shown. The equivalent BASIC line occupies only 18 bytes, 1 for each of the symbols (IF, =, W, AND, and so on), 4 for the line number and 1 for the line delimiter. The more complex the BASIC statement, the more memory overhead there is in the machine code version.

OTHER PLACES TO STORE MACHINE CODE

The main disadvantage with storing code above RAMTOP is that you can't save it. The advantage is that you *can* load other stuff under it. But to save code there are other alternatives.

One place to store the code is in a character string which is easily located via the VARPTR function. ASCII values of the machine code can be appended into the string using the string operator (+). To access the code you can write a BASIC routine, using string functions and a modified version of our loader routine to extract it from the string, decode it from ASCII, load it into RAM and finally run it. The main advantage of using strings is that they can be written on, or read from, a tape (or disk) using BASIC commands. Another place to store code is in DATA statements as we did in the HEXMAS TREES routine. Using this technique, you can save programs on tape and also be protected from wiping out your code with a RUN or a CLEAR.

DEBUGGING

There are no built-in debugging facilities in machine code. Your best bet at this point is to dry-run the routine using old fashioned pencil and

paper. Of course you can insert tracing statements into machine code but watch out for changes in address and jump-sizes.

One useful (though apparently backhanded) trick is to write your routine in BASIC first, and debug *that*: use only BASIC instructions that correspond to machine code (that is *emulate* the machine code in BASIC). It will work slowly, if at all; but it's debuggable.

You may wish to break down altogether and buy an *assembler*. Radio Shack sells a nice one at a reasonable price that has all sorts of nifty debugging aids. Let's try the pencil and paper approach.

DRY-RUNNING A LINE-NUMBERING ROUTINE

As an example of debugging using a dry-run, I'll tell you about my first stab at a routine to renumber the lines of a BASIC program, and how it got debugged. *Being* the first stab, I number the lines 1Ø,2Ø,3Ø,... going up in ten's. Once I got that working, I reckoned I could fancy it up a bit (arbitrary start and finish, arbitrary step size etc.).

Now to do this, you need to know how BASIC lines are stored in memory. They occupy a block of addresses starting at 17385. Each line takes the following form:

NJ	NS	LJ	LS	code for line

Here NS and NJ are the senior and junior bytes of the (in 2-byte hex) address of the next line of BASIC. And since the first two bytes of the next line contain, you guessed it, the address of the next line after that, NS and NJ really contain the address of the next address of the next address... and so on.

LS and LJ are much simpler. They contain the BASIC line numbers we are trying to change.

Our procedure to alter all the line numbers will be a machine language routine and as such it is called, via the USR function, by a BASIC routine. Since, and here's the clever part, the machine routine changes line numbers, it will change those of the BASIC program which has called it!

So if we want to renumber a program we will:

1. load the machine code routine
2. enter the BASIC program to be renumbered
3. add to the end of the BASIC program, using 32768 as a line number, a line containing the USR call to our machine language routine
4. execute the machine language with a GOTO 32768

The machine code will, starting at 17385, read the address of the next BASIC line number, change the contents of the next two bytes, go to the next line and so on until it encounters a one in the senior bit of the LS byte of a line of BASIC.

What kind of a line number does it take to set that senior bit? Well, 32768 or, in binary, 1ØØØØØØØ ØØØØØØØØ, as you can see, contains a set senior bit.

At this point the routine returns control to the BASIC program. Every line has been renumbered except that last one, 32768, which, of course, is only our USR call and can now be deleted as its job is done. Here's the machine code:

```
         LD BC,ØØØA      Ø1 ØA ØØ
LOOP:    LD E,(HL)       5E
         INC HL          23
         LD D,(HL)       56
         PUSH DE         D5
         INC HL twice    23 23
         LD A,(HL)       7E
         BIT7,A          CB 7F
         JRNZ OUT        2Ø ØE
         LD (HL),B       7Ø
         DEC HL          2B
         LD (HL),C       71
         LD H,B          6Ø
         LD L,C          69
         LD DE,ØØØA      11 ØA ØØ
         ADD HL,DE       19
         LD B,H          44
         LD C,L          4Ø
         POP HL          E1
         JR LOOP         18 E7
OUT:     RET             2Ø1
```

Well that was theory. In practice, it led to a decorative crash.

So what was wrong?

To find out, I did a dry-run on a short test program. This is summarized, in a rather cryptic form, below: the idea is to track through step by step and see what happens to the registers, the stack, and the

data in the BASIC area. As you work through, successive lines list how the registers change. You'll need pencil and paper: build up your table yourself, comparing it with mine.

43EB	43EC	43FB	43FC	A	F	B	C	D	E	H	L	STACK
ØØØA	ØØØØ			ØØ	Ø1	ØØ	ØA	43	F9	43	E9	43F9
				ØØ	14	ØØ	ØA	ØØ	ØA	43	EA	
										43	EB	
										43	EC	
										ØØ	ØA	
										ØØ	14	
										43	F9	

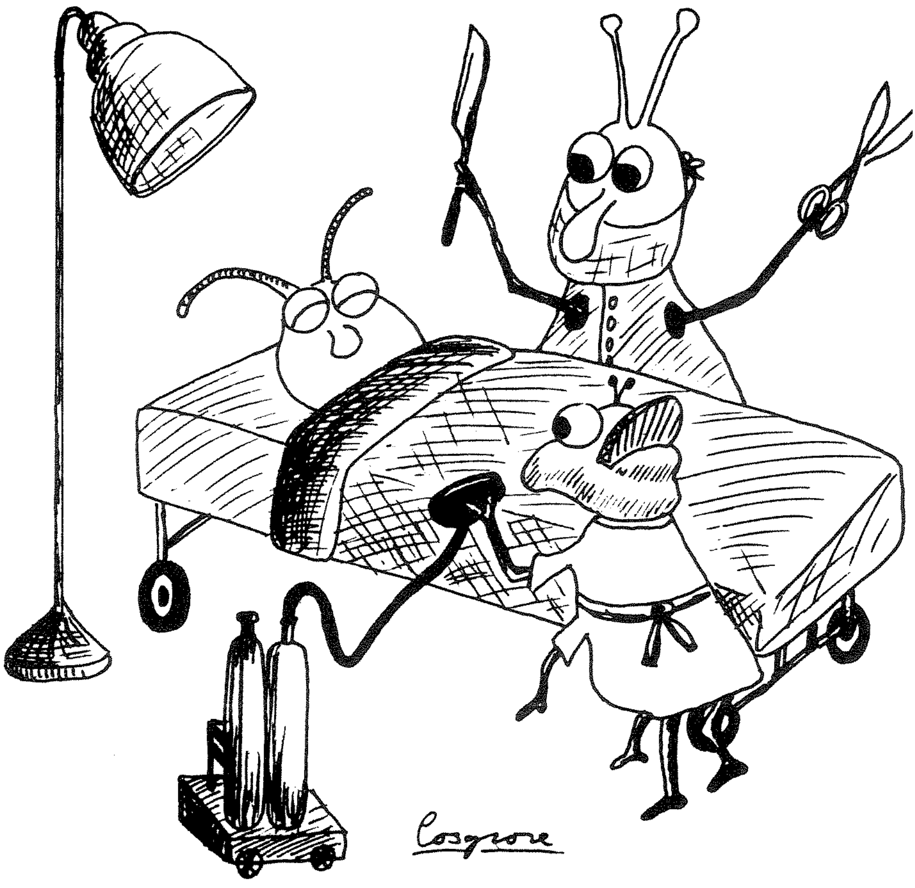
I've assumed we're renumbering a program with two lines of BASIC. Well, not quite. The second line is really our 32768 line that tells us to quit so we've really only got a one line program. Small though it is, it proves to be large enough to locate our bug. Now on to the second line.

43EB	43EC	43FB	43FC	A	F	B	C	D	E	H	L	STACK
ØØØA	ØØØØ			1Ø	ØØ	ØØ	14	44	Ø9	43	FA	4409
										43	FB	

Now we've finished the second line and should be done but notice there is an unPOPed address left in the stack. Here's our problem. Whatever goes onto the stack must come off the stack; otherwise the computer, when it's time to go back to BASIC, will go to the wrong address.

The moral: observe where the error arose. *Not* in the complicated part of the routine, where the main work was done: but in the condition for the work to end. This kind of "boundary problem" is a constant source of errors in programming. Unless you can work out very carefully the precise "edges" of what you are trying to do, you can easily get lost in the wilderness.

Appendices



1 Hex/Decimal Conversion

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
A	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
B	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
C	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
D	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
E	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
F	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

2's complement

ordinary

2 Summary of Z80 Commands

This is a list of all the opcode mnemonics, with a summary of their effects. Those commands explained more fully in the text are given a page reference. The effects on the flags are omitted: for these, consult the books listed in the bibliography under "Machine Code."

ADC	Page 150	Add, including the carry flag. Store in A or HL.
ADD	Page 139	Add, ignoring the carry flag. Store in A or HL.
AND	Page 145	Logic AND on corresponding bits: store in A.
BIT	Page 158	BIT b, r sets the Zero flag according to the value of the b-th bit of the byte in register r. The bits are in the order 76543210 within each byte.
CALL	Page 126	Calls a subroutine. There are conditional calls signalled by the additional letters C (call if the Carry flag is set); M (if the Sign flag is set—"the result (of a compare) is negative"); NC (if the Carry flag is not set); NZ (if the Zero flag is not set); P (if the Sign flag is not set—"the result is positive"); PE (if the Parity flag is set: ignore this one); PO (if the Parity flag is not set: ignore this too); Z(if the Zero flag is set). For flags, see Page 134,156.
CCF		Complement Carry flag (i.e. swap 0 and 1).
CP	Page 146	Compare: sets the flags as if it were a subtraction from A, but leaves A unchanged.
CPD	Page 165	Compare and decrement. Compare through HL; then decrement HL and BC.
CPDR	Page 165	Compare, decrement, repeat: block search. Like CPD but repeating until either the result of the comparison is 0, or BC reaches 0.
CPI	Page 165	Like CPD except that HL increments; BC still decrements.
CPIR	Page 165	Like CPDR but incrementing HL.
CPL		Complement (flip bits of) the A-register.
DAA		Decimal adjust accumulator. Used in binary-coded decimal work: ignore.
DEC	Page 148	Decrement: reduce value by 1.
DI		Disable interrupts. Ignore.
DJNZ	Page 149	Decrement, jump if non-zero. Decrement B and jump relative unless the Zero flag is set. Used in loops like a BASIC FOR/NEXT.
EI		Enable interrupts. Ignore.
EX	Page 169	Exchange values. Instructions with (SP) exchange the registers HL, IX or IY with the top of the stack.
EXX	Page 169	Exchange all three register-pairs BC, DE, HL with their alternates BC', DE', HL'.
HALT		Wait for an interrupt. Unless you've got hardware attached, and know what you're doing, DO NOT USE as the program will wait forever.
IM		Interrupt mode: ignore.
IN		Input from a device. Ignore.
INC	Page 148	Increment: increase value by 1.
IND, INDR, INI, INIR		Input commands analogous to LDD, LDDR, LDI, LDIR. Ignore.
JP	Page 146	Jump. Variants with added C, M, NC, NZ, P, PE, PO, Z are conditional jumps, with conditions as for CALL.

JR	Page 146	Jump relative—followed by a 1-byte displacement. Conditional variants are C, NC, NZ, Z only.
LD	Page 136	Load. Can use all five addressing modes.
LDD		Not the same as LD D! Load what HL points to into what DE points to: decrement BC, DE, HL.
LDDR	Page 166	Load, decrement, repeat: block transfer. Do LDD until BC hits zero. Copies a block of memory whose length is stored in BC, out of what HL points to and into what DE points to.
LDI	Page 166	Like LDD except that HL and DE increment: BC still decrements.
LDIR	Page 166	Like LDDR except that HL and DE increment.
NEG		Negative: change the sign of the contents of A.
NOP		No operation. Do nothing for 1 time-cycle—i.e. waste time. Useful for temporary deletion of instructions when debugging: harmless and helpful.
OR	Page 145	Logic OR on bits. Store in A.
OTDR, OTIR, OUT, OUTD, OUTI		Various outputs. Ignore.
POP	Page 151	Pop from stack into indicated register.
PUSH	Page 151	Push from register on to stack.
RES		Reset a bit—i.e. make it zero.
RET	Page 126	Return from subroutine. Conditional returns, corresponding to the possibilities for CALL, are possible. (Conditions on a CALL need not match those on a RET!)
RETI, RETN		Return from interrupt subroutines. Ignore.
RL		Rotate left: like a shift, except that the carry flag is included as if it were bit number 8.
RLA		Rotate left accumulator. Like RL A but with a different effect on the flags.
RLC		Not the same as RL C! Rotate left, but put bit 7 into carry *and* into bit Ø.
RLCA		Like RLC A, but same flag difference as RLA.
RLD		Not what you'd expect at all: rotate left decimal. Used for binary coded decimal: ignore.
RR		Like RL but to the right.
RRA		Like RLA.
RRC		Like RLC.
RRCA		Like RLCA.
RRD		Like RLD.
RST		Like CALL, but only from addresses Ø, 8, 1Ø, 18, 2Ø, 28, 3Ø, 38 (hex). These are all in the ROM on the ZX81: see Ian Logan's books in the Bibliography. RST Ø is like temporarily disconnecting the power.
SBC	Page 150	Subtract, taking account of the carry flag. Store in A or HL.
SCF		Set carry flag (to 1).
SET		Set a bit—i.e. make it 1.
SLA	Page 150	Shift left arithmetic. All bits move up 1; bit Ø becomes Ø.
SRA	Page 150	Shift right arithmetic. Move bits down 1; copy bit 7 into 6 *and* 7.
SRL	Page 150	Shift right logical. Move bits down 1 place; make bit 7 zero.
SUB	Page 139	Subtract, ignoring carry. Store in A. (There is no SUB HL, r command: if you want one, reset the Carry flag and use SBC.
XOR	Page 146	Exclusive or on each bit. Store in A.

3 Z80 Opcodes

Examples:

LD BC, nn has the opcode Ø1 nn, so LD BC, 732F codes as Ø1 2F 73.

LD A, (IY + d) " " " FD 7E d, so LD A (IY + Ø7) " " FD 7E Ø7.

The table of opcodes is based on one published by Zilog Inc.

ADC A, (HL)	8E
ADC A, (IX + d)	DD8Ed
ADC A, (IY + d)	FD8Ed
ADC A, A	8F
ADC A, B	88
ADC A, C	89
ADC A, D	8A
ADC A, E	8B
ADC A, H	8C
ADC A, L	8D
ADC A, n	CEn
ADC HL, BC	ED4A
ADC HL, DE	ED5A
ADC HL, HL	ED6A
ADC HL, SP	ED7A
ADD A, (HL)	86
ADD A, (IX + d)	DD86d
ADD A, (IY + d)	FD86d
ADD A, A	87
ADD A, B	80
ADD A, C	81
ADD A,D	82
ADD A, E	83
ADD A, H	84
ADD A, L	85
ADD A, n	C6n
ADD HL, BC	09
ADD HL, DE	19
ADD HL, HL	29
ADD HL, SP	39
ADD IX, BC	DD09
ADD IX, DE	DD19
ADD IX, IX	DD29
ADD IX, SP	DD39
ADD IY, BC	FD09
ADD IY, DE	FD19
ADD IY, IY	FD29
ADD IY, SP	FD39
AND (HL)	A6
AND (IX + d)	DDA6d
AND (IY + d)	FDA6d
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND n	E6n
BIT 0, (HL)	CB46
BIT 0, (IX + d)	DDCBd46
BIT 0, (IY + d)	FDCBd46
BIT 0, A	CB47
BIT 0, B	CB40
BIT 0, C	CB41
BIT 0, D	CB42
BIT 0, E	CB43
BIT 0, H	CB44
BIT 0, L	CB45
BIT 1, (HL)	CB4E
BIT 1, (IX + d)	DDCBd4E
BIT 1, (IY + d)	FDCBd4E
BIT 1, A	CB4F
BIT 1, B	CB48
BIT 1, C	CB49
BIT 1, D	CB4A
BIT 1, E	CB4B
BIT 1, H	CB4C
BIT 1, L	CB4D
BIT 2, (HL)	CB56
BIT 2, (IX + d)	DDCBd56
BIT 2, (IY + d)	FDCBd56
BIT 2, A	CB57
BIT 2, B	CB50
BIT 2, C	CB51
BIT 2, D	CB52
BIT 2, E	CB53
BIT 2, H	CB54
BIT 2, L	CB55
BIT 3, (HL)	CB5E
BIT 3, (IX + d)	DDCBd5E
BIT 3, (IY + d)	FDCBd5E
BIT 3, A	CB5F
BIT 3, B	CB58
BIT 3, C	CB59
BIT 3, D	CB5A
BIT 3, E	CB5B
BIT 3, H	CB5C
BIT 3, L	CB5D
BIT 4, (HL)	CB66
BIT 4, (IX + d)	DDCBd66
BIT 4, (IY + d)	FDCBd66
BIT 4, A	CB67
BIT 4, B	CB60
BIT 4, C	CB61
BIT 4, D	CB62
BIT 4, E	CB63
BIT 4, H	CB64
BIT 4, L	CB65
BIT 5, (HL)	CB6E
BIT 5, (IX + d)	DDCBd6E
BIT 5, (IY + d)	FDCBd6E
BIT 5, A	CB6F
BIT 5, B	CB68
BIT 5, C	CB69
BIT 5, D	CB6A
BIT 5, E	CB6B
BIT 5, H	CB6C
BIT 5, L	CB6D
BIT 6, (HL)	CB76
BIT 6, (IX + d)	DDCBd76
BIT 6, (IY + d)	FDCBd76
BIT 6, A	CB77
BIT 6, B	CB70
BIT 6, C	CB71
BIT 6, D	CB72
BIT 6, E	CB73
BIT 6, H	CB74
BIT 6, L	CB75
BIT 7, (HL)	CB7E
BIT 7, (IX + d)	DDCBd7E
BIT 7, (IY + d)	FDCBd7E
BIT 7, A	CB7F
BIT 7, B	CB78
BIT 7, C	CB79
BIT 7, D	CB7A
BIT 7, E	CB7B
BIT 7, H	CB7C
BIT 7, L	CB7D
CALL C, nn	DCnn
CALL M, nn	FCnn
CALL NC, nn	D4nn
CALL nn	CDnn
CALL NZ, nn	C4nn
CALL P, nn	F4nn
CALL PE, nn	ECnn
CALL PO, nn	E4nn
CALL Z, nn	CCnn
CCF	3F
CP (HL)	BE
CP (IX + d)	DDBEd
CP (IY + d)	FDBEd
CPA	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP n	FEn
CPD	EDA9
CPDR	EDB9
CPI	EDA1
CPIR	EDB1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX + d)	DD35d
DEC (IY + d)	FD35d

DEC A	3D
DEC B	05
DEC BC	0B
DEC C	0D
DEC D	15
DEC DE	1B
DEC E	1D
DEC H	25
DEC HL	2B
DEC IX	DD2B
DEC IY	FD2B
DEC L	2D
DEC SP	3B
DI	F3
DJNZ, d	10d
EI	FB
EX (SP), HL	E3
EX (SP), IX	DDE3
EX (SP), IY	FDE3
EX AF, AF'	08
EX DE, HL	EB
EXX	D9
HALT	76
IM 0	ED46
IM 1	ED56
IM 2	ED5E
IN A, (C)	ED78
IN A, (n)	DBn
IN B, (C)	ED40
IN C, (C)	ED48
IN D, (C)	ED50
IN E, (C)	ED58
IN H, (C)	ED60
IN L, (C)	ED68
INC (HL)	34
INC (IX + d)	DD34d
INC (IY + d)	FD34d
INC A	3C
INC B	04
INC BC	03
INC C	0C
INC D	14
INC DE	13
INC E	1C
INC H	24
INC HL	23
INC IX	DD23
INC IY	FD23
INC L	2C
INC SP	33
IND	EDAA
INDR	EDBA
INI	EDA2
INIR	EDB2
JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9
JP C, nn	DAnn
JP M, nn	FAnn
JP NC, nn	D2nn
JP nn	C3nn
JP NZ, nn	C2nn
JP P, nn	F2nn
JP PE, nn	EAnn
JP PO, nn	E2nn
JP Z, nn	CAnn
JR C, d	38d
JR, d	18d
JR NC, d	30d
JR NZ, d	20d
JR Z, d	28d
LD (BC), A	02
LD (DE), A	12
LD (HL), A	77
LD (HL), B	70
LD (HL), C	71
LD (HL), D	72
LD (HL), E	73
LD (HL), H	74
LD (HL), L	75
LD (HL), n	36n
LD (IX + d), A	DD77d
LD (IX + d), B	DD70d
LD (IX + d), C	DD71d
LD (IX + d), D	DD72d
LD (IX + d), E	DD73d
LD (IX + d), H	DD74d
LD (IX + d), L	DD75d
LD (IX + d), n	DD36dn
LD (IY + d), A	FD77d
LD (IY + d), B	FD70d
LD (IY + d), C	FD71d
LD (IY + d), D	FD72d
LD (IY + d), E	FD73d
LD (IY + d), H	FD74d
LD (IY + d), L	FD75d
LD (IY + d), n	FD36dn
LD (nn), A	32nn
LD (nn), BC	ED43nn
LD (nn), DE	ED53nn
LD (nn), HL	22nn
LD (nn), IX	DD22nn
LD (nn), IY	FD22nn
LD (nn), SP	ED73nn
LD A, (BC)	0A
LD A, (DE)	1A
LD A, (HL)	7E
LD A, (IX + d)	DD7Ed
LD A, (IY + d)	FD7Ed
LD A, (nn)	3Ann
LD A, A	7F
LD A, B	78
LD A, C	79
LD A, D	7A
LD A, E	7B
LD A, H	7C
LD A, I	ED57
LD A, L	7D
LD A, n	3En
LD B, (HL)	46
LD B, (IX + d)	DD46d
LD B, (IY + d)	FD46d
LD B, A	47
LD B, B	40
LD B, C	41
LD B, D	42
LD B, E	43
LD B, H	44
LD B, L	45
LD B, n	06n
LD BC, (nn)	ED4Bnn
LD BC, nn	01nn
LD C, (HL)	4E
LD C, (IX + d)	DD4Ed
LD C, (IY + d)	FD4Ed
LD C, A	4F
LD C, B	48
LD C, C	49
LD C, D	4A
LD C, E	4B
LD C, H	4C
LD C, L	4D
LD C, n	0En
LD D, (HL)	56
LD D, (IX + d)	DD56d
LD D, (IY + d)	FD56d
LD D, A	57
LD D, B	50
LD D, C	51
LD D, D	52
LD D, E	53
LD D, H	54
LD D, L	55
LD D, n	16n
LD DE, (nn)	ED5Bnn
LD DE, nn	11nn
LD E, (HL)	5E
LD E, (IX + d)	DD5Ed
LD E, (IY + d)	FD5Ed
LD E, A	5F
LD E, B	58
LD E, C	59
LD E, D	5A
LD E, E	5B
LD E, H	5C
LD E, L	5D
LD E, n	1En
LD H, (HL)	66
LD H, (IX + d)	DD66d
LD H, (IY + d)	FD66d
LD H, A	67
LD H, B	60
LD H, C	61
LD H, D	62
LD H, E	63
LD H, H	64
LD H, L	65
LD H, n	26n
LD HL, (nn)	2Ann
LD HL, nn	21nn
LD I, A	ED47
LD IX, (nn)	DD2Ann
LD IX, nn	DD21nn
LD IY, (nn)	FD2Ann
LD IY, nn	FD21nn
LD L, (HL)	6E
LD L, (IX + d)	DD6Ed
LD L, (IY + d)	FD6Ed
LD L, A	6F
LD L, B	68
LD L, C	69
LD L, D	6A
LD L, E	6B
LD L, H	6C
LD L, L	6D
LD L, n	2En
LD SP, (nn)	ED7Bnn
LD SP, HL	F9
LD SP, IX	DDF9
LD SP, IY	FDF9
LD SP, nn	31nn
LDD	EDA8
LDDR	EDB8
LDI	EDA0
LDIR	EDB0
NEG	ED44
NOP	00
OR (HL)	B6
OR (IX + d)	DDB6d
OR (IY + d)	FDB6d
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6n
OTDR	EDBB
OTIR	EDB3
OUT (C), A	ED79
OUT (C), B	ED41
OUT (C), C	ED49
OUT (C), D	ED51
OUT (C), E	ED59
OUT (C), H	ED61
OUT (C), L	ED69
OUT (n), A	D3n

OUTD	EDAB
OUTI	EDA3
POPAF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
RES 0, (HL)	CB86
RES 0, (IX + d)	DDCBd86
RES 0, (IY + d)	FDCBd86
RES 0, A	CB87
RES 0, B	CB80
RES 0, C	CB81
RES 0, D	CB82
RES 0, E	CB83
RES 0, H	CB84
RES 0, L	CB85
RES 1, (HL)	CB8E
RES 1, (IX + d)	DDCBd8E
RES 1, (IY + d)	FDCBd8E
RES 1, A	CB8F
RES 1, B	CB88
RES 1, C	CB89
RES 1, D	CB8A
RES 1, E	CB8B
RES 1, H	CB8C
RES 1, L	CB8D
RES 2, (HL)	CB96
RES 2, (IX + d)	DDCBd96
RES 2, (IY + d)	FDCBd96
RES 2, A	CB97
RES 2, B	CB90
RES 2, C	CB91
RES 2, D	CB92
RES 2, E	CB93
RES 2, H	CB94
RES 2, L	CB95
RES 3, (HL)	CB9E
RES 3, (IX + d)	DDCBd9E
RES 3, (IY + d)	FDCBd9E
RES 3, A	CB9F
RES 3, B	CB98
RES 3, C	CB99
RES 3, D	CB9A
RES 3, E	CB9B
RES 3, H	CB9C
RES 3, L	CB9D
RES 4, (HL)	CBA6
RES 4, (IX + d)	DDCBdA6
RES 4, (IY + d)	FDCBdA6
RES 4, A	CBA7
RES 4, B	CBA0
RES 4, C	CBA1
RES 4, D	CBA2
RES 4, E	CBA3
RES 4, H	CBA4
RES 4, L	CBA5
RES 5, (HL)	CBAE
RES 5, (IX + d)	DDCBdAE
RES 5, (IY + d)	FDCBdAE
RES 5, A	CBAF
RES 5, B	CBA8
RES 5, C	CBA9
RES 5, D	CBAA
RES 5, E	CBAB
RES 5, H	CBAC
RES 5, L	CBAD
RES 6, (HL)	CBB6
RES 6, (IX + d)	DDCBdB6
RES 6, (IY + d)	FDCBdB6
RES 6, A	CBB7
RES 6, B	CBB0
RES 6, C	CBB1
RES 6, D	CBB2
RES 6, E	CBB3
RES 6, H	CBB4
RES 6, L	CBB5
RES 7, (HL)	CBBE
RES 7, (IX + d)	DDCBdBE
RES 7, (IY + d)	FDCBdBE
RES 7, A	CBBF
RES 7, B	CBB8
RES 7, C	CBB9
RES 7, D	CBBA
RES 7, E	CBBB
RES 7, H	CBBC
RES 7, L	CBBD
RET	C9
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET P	F0
RET PE	E8
RET PO	E0
RET Z	C8
RETI	ED4D
RETN	ED45
RL (HL)	CB16
RL (IX + d)	DDCBd16
RL (IY + d)	FDCBd16
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RLA	17
RLC (HL)	CB06
RLC (IX + d)	DDCBd06
RLC (IY + d)	FDCBd06
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLCA	07
RLD	ED6F
RR (HL)	CB1E
RR (IX + d)	DDCBd1E
RR (IY + d)	FDCBd1E
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RRA	1F
RRC (HL)	CB0E
RRC (IX + d)	DDCBd0E
RRC (IY + d)	FDCBd0E
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC A	0F
RRD	ED67
RST 0	C7
RST 10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8	CF
SBC A, (HL)	9E
SBC A, (IX + d)	DD9Ed
SBC A, (IY + d)	FD9Ed
SBC A, A	9F
SBC A, B	98
SBC A, C	99
SBC A, D	9A
SBC A, E	9B
SBC A, H	9C
SBC A, L	9D
SBC A, n	DEn
SBC HL, BC	ED42
SBC HL, DE	ED52
SBC HL, HL	ED62
SBC HL, SP	ED72
SCF	37
SET 0, (HL)	CBC6
SET 0, (IX + d)	DDCBdC6
SET 0, (IY + d)	FDCBdC6
SET 0, A	CBC7
SET 0, B	CBC0
SET 0, C	CBC1
SET 0, D	CBC2
SET 0, E	CBC3
SET 0, H	CBC4
SET 0, L	CBC5
SET 1, (HL)	CBCE
SET 1, (IX + d)	DDCBdCE
SET 1, (IY + d)	FDCBdCE
SET 1, A	CBCF
SET 1, B	CBC8
SET 1, C	CBC9
SET 1, D	CBCA
SET 1, E	CBCB
SET 1, H	CBCC
SET 1, L	CBCD
SET 2, (HL)	CBD6
SET 2, (IX + d)	DDCBdD6
SET 2, (IY + d)	FDCBdD6
SET 2, A	CBD7
SET 2, B	CBD0
SET 2, C	CBD1
SET 2, D	CBD2
SET 2, E	CBD3
SET 2, H	CBD4
SET 2, L	CBD5
SET 3, B	CBD8
SET 3, (HL)	CBDE
SET 3, (IX + d)	DDCBdDE
SET 3, (IY + d)	FDCBdDE
SET 3, A	CBDF
SET 3, C	CBD9
SET 3, D	CBDA
SET 3, E	CBDB
SET 3, H	CBDC
SET 3, L	CBDD
SET 4, (HL)	CBE6
SET 4,(IX + d)	DDCBdE6
SET 4, (IY + d)	FDCBdE6
SET 4, A	CBE7
SET 4, B	CBE0
SET 4, C	CBE1
SET 4, D	CBE2
SET 4, E	CBE3
SET 4, H	CBE4
SET 4, L	CBE5
SET 5, (HL)	CBEE
SET 5, (IX + d)	DDCBdEE
SET 5, (IY + d)	FDCBdEE

ET 5, A	CBEF
ET 5, B	CBE8
ET 5, C	CBE9
ET 5, D	CBEA
ET 5, E	CBEB
ET 5, H	CBEC
ET 5, L	CBED
ET 6, (HL)	CBF6
ET 6, (IX + d)	DDCBdF6
ET 6, (IY + d)	FDCBdF6
ET 6, A	CBF7
ET 6, B	CBF0
ET 6, C	CBF1
ET 6, D	CBF2
ET 6, E	CBF3
ET 6, H	CBF4
ET 6, L	CBF5
ET 7, (HL)	CBFE
ET 7, (IX + d)	DDCBdFE
ET 7, (IY + d)	FDCBdFE
ET 7, A	CBFF
ET 7, B	CBF8
ET 7, C	CBF9
ET 7, D	CBFA
ET 7, E	CBFB
ET 7, H	CBFC
ET 7, L	CBFD
SLA (HL)	CB26
SLA (IX + d)	DDCBd26
SLA (IY + d)	FDCBd26
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SRA (HL)	CB2E
SRA (IX + d)	DDCBd2E
SRA (IY + d)	FDCBd2E
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRL (HL)	CB3E
SRL (IX + d)	DDCBd3E
SRL (IY + d)	FDCBd3E
SRL A	CB3F
SRL B	CB38
SRL C	CB39
SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SUB (HL)	96
SUB (IX + d)	DD96d
SUB (IY + d)	FD96d
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6n
XOR (HL)	AE
XOR (IX + d)	DDAEd
XOR (IY + d)	FDAEd
XOR A	AF
XOR B	A8
XOR C	A9
XOR D	AA
XOR E	AB
XOR H	AC
XOR L	AD
XOR n	EEn

Bibliography

Data structures

Aho, Hopcroft and Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley.
Berztiss, *Data Structures, Theory and Practice,* Academic Press.
Brillinger and Cohen, *Introduction to Data Structures and Non-numeric Computation,* Prentice Hall.

Machine code

Carr, *Z80 User's Manual,* Boston Publishing Co. Inc.
Nichols, Nichols and Rony *Z80 Microprocessor Programming and Interfacing,* Howard Sams & Co.
Spracklen, *Z80 and 8080 Assembly Language Programming,* Hayden.
Zaks, *Programming the Z80, Sybex.*
Zilog Z80 CPU Programming Reference Card.
Zilog Z80 CPU Technical Manual

General

Brady, *The Theory of Computer Science,* Chapman and Hall.
Dahl, Dijksta, and Hoare, *Structured Programming,* Academic Press.
Sloan, *Introduction to Minicomputers and Microcomputers,* Addison-Wesley.
Tocher, *The Art of Simulation,* E.U.P.
Wegner, *Programming Languages, Information Structures, and Machine Organization.* McGraw-Hill.
Weizenbaum, *Computer Power and Human Reason,* W.H. Freeman.

Index

TRS-80 Model III or Model IV users: Are you taking full advantage of your sophisticated equipment? Have you mastered BASIC, and now long to break free of "canned" software? Are you anxious to learn the tricks of serious programming? If so, this book was written just for you.

CONTROL YOUR TRS-80 will teach you the techniques for writing more creative programs, clearly and quickly. It will reveal the secrets of data structures, and introduce Machine Code, the computer's internal language. Knowing Machine Code gives you the advantage of getting into the inner workings of your computer, and allows you to control the speed and complexity of your programs. This book does not train you with rote exercises; it teaches the concepts of professional programming.

With the help of this book, you can use your TRS-80 to:

- Search an inventory list
- Push onto and pop off a stack
- Catalogue your library
- Build a game tree
- Program a simple word processor
- Perform a real-life simulation study

Machine Code instructions enable you to:

- Display a checkerboard pattern (instantly)
- Draw a "Hexmas" tree
- Add and multiply numbers
- Move data around in RAM
- Take advantage of split-screen capability
- Scroll sideways
- Renumber BASIC lines automatically

Plus a sample "game" program to test your French vocabulary, and excellent appendices for reference.

In short, all you need to CONTROL YOUR TRS-80!

CONTROL YOUR TRS-80

ISBN 3-7643-3143-7 $14.95